O'REILLY®

TURING 图灵程序设计丛书

第2版

Python测试驱动开发

使用Django、Selenium和JavaScript进行Web编程

Test-Driven Development with Python



[英] 哈利·J.W. 帕西瓦尔 著

安道 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

安道

专注于现代计算机技术的自由翻译，译有《流畅的 Python》《Flask Web 开发》《Ruby on Rails 教程》等。

数字版权声明

TURING 图灵程序设计丛书

Python测试驱动开发：使用Django、Selenium和JavaScript进行Web编程（第2版）

Test-Driven Development with Python, Second Edition

[英] 哈利·J.W. 帕西瓦尔 著
安道 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北京

图书在版编目（C I P）数据

Python测试驱动开发 : 使用Django、Selenium和JavaScript进行Web编程 : 第2版 / (英) 哈利·帕西瓦尔 (Harry J. W. Percival) 著 ; 安道译. -- 北京 : 人民邮电出版社, 2018.7
（图灵程序设计丛书）
ISBN 978-7-115-48557-1

Ⅰ. ①P… Ⅱ. ①哈… ②安… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第111743号

内 容 提 要

本书从最基础的知识开始，讲解 Web 开发的整个流程，展示如何使用 Python 做测试驱动开发。本书由三个部分组成。第一部分介绍了测试驱动开发和 Django 的基础知识，并在每个阶段进行严格的单元测试。第二部分讨论了 Web 开发要素，探讨了 Web 开发过程中不可避免的问题，以及如何通过测试解决这些问题。第三部分探讨了一些高级话题，如模拟技术、集成第三方认证系统、Ajax、测试固件以及持续集成等。

第 2 版全部使用 Python 3，并针对新版 Django 全面升级，介绍了由外而内的测试驱动开发流程。

本书适合 Web 开发人员阅读。

◆ 著　　　　[英] 哈利·J.W. 帕西瓦尔
　译　　　　安　道
　责任编辑　朱　巍
　执行编辑　夏静文
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　印刷

◆ 开本：800×1000　1/16
　印张：30.25
　字数：721千字　　　　2018年 7 月第 1 版
　印数：1 – 3 000册　　　2018年 7 月北京第 1 次印刷
　著作权合同登记号　图字：01-2018-3460号

定价：119.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言……xv

准备工作和应具备的知识……xxi

配套视频……xxviii

致谢……xxix

第一部分　TDD 和 Django 基础

第 1 章　使用功能测试协助安装 Django……2

1.1　遵从测试山羊的教诲，没有测试什么也别做……2
1.2　让Django运行起来……4
1.3　创建Git仓库……6

第 2 章　使用 `unittest` 模块扩展功能测试……10

2.1　使用功能测试驱动开发一个最简可用的应用……10
2.2　Python标准库中的`unittest`模块……12
2.3　提交……14

第 3 章　使用单元测试测试简单的首页……16

3.1　第一个Django应用，第一个单元测试……16
3.2　单元测试及其与功能测试的区别……17
3.3　Django中的单元测试……18
3.4　Django中的MVC、URL和视图函数……19
3.5　终于可以编写一些应用代码了……20
3.6　urls.py……22
3.7　为视图编写单元测试……23

第 4 章　测试（及重构）的目的······28
4.1　编程就像从井里打水······28
4.2　使用Selenium测试用户交互······30
4.3　遵守“不测试常量”规则，使用模板解决这个问题······32
4.3.1　使用模板重构······33
4.3.2　Django测试客户端······35
4.4　关于重构······37
4.5　接着修改首页······38
4.6　总结：TDD流程······39
第 5 章　保存用户输入：测试数据库······42
5.1　编写表单，发送POST请求······42
5.2　在服务器中处理POST请求······45
5.3　把Python变量传入模板中渲染······46
5.4　事不过三，三则重构······50
5.5　Django ORM和第一个模型······51
5.5.1　第一个数据库迁移······53
5.5.2　测试向前走得挺远······53
5.5.3　添加新字段就要创建新迁移······54
5.6　把POST请求中的数据存入数据库······55
5.7　处理完POST请求后重定向······57
5.8　在模板中渲染待办事项······59
5.9　使用迁移创建生产数据库······61
5.10　回顾······64
第 6 章　改进功能测试：确保隔离，去掉含糊的休眠······66
6.1　确保功能测试之间相互隔离······66
6.2　升级Selenium和Geckodriver······70
6.3　隐式等待、显式等待和含糊的`time.sleep`······70
第 7 章　步步为营······75
7.1　必要时做少量的设计······75
7.1.1　不要预先做大量设计······75
7.1.2　YAGNI······76
7.1.3　REST（式）······76
7.2　使用TDD实现新设计······77
7.3　确保出现回归测试······78
7.4　逐步迭代，实现新设计······80
7.5　自成一体的第一步：新的URL······81
7.5.1　一个新URL······82

7.5.2 一个新视图函数 82
7.6 变绿了吗？该重构了 84
7.7 再迈一小步：一个新模板，用于查看清单 84
7.8 第三小步：用于添加待办事项的URL 86
7.8.1 用来测试新建清单的测试类 87
7.8.2 用于新建清单的URL和视图 88
7.8.3 删除当前多余的代码和测试 89
7.8.4 出现回归！让表单指向刚添加的新URL 89
7.9 下定决心，调整模型 90
7.9.1 外键关系 92
7.9.2 根据新模型定义调整其他代码 93
7.10 每个列表都应该有自己的URL 95
7.10.1 捕获URL中的参数 96
7.10.2 按照新设计调整new_list视图 97
7.11 功能测试又检测到回归 98
7.12 还需要一个视图，把待办事项加入现有清单 99
7.12.1 小心霸道的正则表达式 99
7.12.2 最后一个新URL 100
7.12.3 最后一个新视图 101
7.12.4 直接测试响应上下文对象 102
7.13 使用URL引入做最后一次重构 103

第二部分　Web 开发要素

第 8 章　美化网站：布局、样式及其测试方法 108
8.1 如何在功能测试中测试布局和样式 108
8.2 使用CSS框架美化网站 111
8.3 Django模板继承 112
8.4 集成Bootstrap 114
8.5 Django中的静态文件 115
8.6 使用Bootstrap中的组件改进网站外观 117
8.6.1 超大文本块 118
8.6.2 大型输入框 118
8.6.3 样式化表格 118
8.7 使用自己编写的CSS 118
8.8 补遗：collectstatic命令和其他静态目录 120
8.9 没谈到的话题 122

第 9 章　使用过渡网站测试部署 123
9.1　TDD以及部署的危险区域 124
9.2　一如既往，先写测试 125
9.3　注册域名 127
9.4　手动配置托管网站的服务器 127
9.4.1　选择在哪里托管网站 127
9.4.2　搭建服务器 128
9.4.3　用户账户、SSH和权限 128
9.4.4　安装Nginx 128
9.4.5　安装Python 3.6 129
9.4.6　解析过渡环境和线上环境所用的域名 130
9.4.7　使用功能测试确认域名可用而且Nginx正在运行 130
9.5　手动部署代码 130
9.5.1　调整数据库的位置 131
9.5.2　手动创建虚拟环境，使用requirements.txt 133
9.5.3　简单配置Nginx 134
9.5.4　使用迁移创建数据库 136
9.6　手动部署大功告成 137
第 10 章　为部署到生产环境做好准备 139
10.1　换用Gunicorn 139
10.2　让Nginx伺服静态文件 140
10.3　换用Unix套接字 141
10.4　把DEBUG设为False，设置ALLOWED_HOSTS 142
10.5　使用Systemd确保引导时启动Gunicorn 143
10.6　考虑自动化 144
10.7　保存进度 147
第 11 章　使用 Fabric 自动部署 148
11.1　分析一个Fabric部署脚本 149
11.1.1　分析一个Fabric部署脚本 149
11.1.2　使用Git拉取源码 150
11.1.3　更新settings.py 151
11.1.4　更新虚拟环境 151
11.1.5　需要时迁移数据库 152
11.2　试用部署脚本 152
11.2.1　部署到线上服务器 154
11.2.2　使用sed配置Nginx和Gunicorn 155
11.3　使用Git标签标注发布状态 157
11.4　延伸阅读 157

第 12 章 输入验证和测试的组织方式……159
12.1 针对验证的功能测试：避免提交空待办事项……159
12.1.1 跳过测试……160
12.1.2 把功能测试分拆到多个文件中……161
12.1.3 运行单个测试文件……163
12.2 功能测试新工具：通用显式等待辅助方法……164
12.3 补完功能测试……167
12.4 重构单元测试，分拆成多个文件……168
第 13 章 数据库层验证……171
13.1 模型层验证……172
13.1.1 self.assertRaises上下文管理器……172
13.1.2 Django怪异的表现：保存时不验证数据……173
13.2 在视图中显示模型验证错误……173
13.3 Django模式：在渲染表单的视图中处理POST请求……177
13.3.1 重构：把new_item实现的功能移到view_list中……178
13.3.2 在view_list视图中执行模型验证……180
13.4 重构：去除硬编码的URL……182
13.4.1 模板标签{% url %}……182
13.4.2 重定向时使用get_absolute_url……183
第 14 章 简单的表单……186
14.1 把验证逻辑移到表单中……186
14.1.1 使用单元测试探索表单API……187
14.1.2 换用Django中的ModelForm类……188
14.1.3 测试和定制表单验证……189
14.2 在视图中使用这个表单……191
14.2.1 在处理GET请求的视图中使用这个表单……191
14.2.2 大量查找和替换……192
14.3 在处理POST请求的视图中使用这个表单……194
14.3.1 修改new_list视图的单元测试……195
14.3.2 在视图中使用这个表单……196
14.3.3 使用这个表单在模板中显示错误消息……196
14.4 在其他视图中使用这个表单……197
14.4.1 定义辅助方法，简化测试……197
14.4.2 意想不到的好处：HTML5自带的客户端验证……199
14.5 值得鼓励……201
14.6 这难道不是浪费时间吗……201
14.7 使用表单自带的save方法……202

第 15 章　高级表单 ······ 205
15.1　针对重复待办事项的功能测试 ······ 205
15.1.1　在模型层禁止重复 ······ 206
15.1.2　题外话：查询集合排序和字符串表示形式 ······ 208
15.1.3　重写旧模型测试 ······ 210
15.1.4　保存时确实会显示完整性错误 ······ 211
15.2　在视图层试验待办事项重复验证 ······ 212
15.3　处理唯一性验证的复杂表单 ······ 213
15.4　在清单视图中使用`ExistingListItemForm` ······ 215
15.5　小结：目前所学的Django测试知识 ······ 217
第 16 章　试探 JavaScript ······ 219
16.1　从功能测试开始 ······ 219
16.2　安装一个基本的JavaScript测试运行程序 ······ 221
16.3　使用jQuery和`<div>`固件元素 ······ 223
16.4　为想要实现的功能编写JavaScript单元测试 ······ 225
16.5　固件、执行顺序和全局状态：JavaScript测试的重大挑战 ······ 227
16.5.1　使用`console.log`打印调试信息 ······ 227
16.5.2　使用初始化函数精确控制执行时间 ······ 229
16.6　经验做法：`onload`样板代码和命名空间 ······ 230
16.7　JavaScript测试在TDD循环中的位置 ······ 232
16.8　一些缺憾 ······ 232
第 17 章　部署新代码 ······ 234
17.1　部署到过渡服务器 ······ 234
17.2　部署到线上服务器 ······ 235
17.3　如果看到数据库错误该怎么办 ······ 235
17.4　总结：为这次新发布打上Git标签 ······ 235

第三部分　高级话题

第 18 章　用户身份验证、探究及去掉探究代码 ······ 238
18.1　无密码验证 ······ 238
18.2　探索性编程（又名“探究”） ······ 239
18.2.1　为此次探究新建一个分支 ······ 239
18.2.2　前端登录UI ······ 240
18.2.3　从Django中发出邮件 ······ 240
18.2.4　使用环境变量，避免源码中出现机密信息 ······ 242
18.2.5　在数据库中存储令牌 ······ 243
18.2.6　自定义身份验证模型 ······ 243
18.2.7　结束自定义Django身份验证功能 ······ 244

18.3 去掉探究代码……248
18.4 一个极简的自定义用户模型……251
18.5 令牌模型：把电子邮件地址与唯一的ID关联起来……254
第 19 章 使用驭件测试外部依赖或减少重复……257
19.1 开始之前布好基本管道……257
19.2 自己动手模拟（打猴子补丁）……258
19.3 Python的模拟库……261
19.3.1 使用unittest.patch……261
19.3.2 让测试向前迈一小步……263
19.3.3 测试Django消息框架……263
19.3.4 在HTML中添加消息……265
19.3.5 构建登录URL……266
19.3.6 确认给用户发送了带有令牌的链接……267
19.4 去除自定义的身份验证后端中的探究代码……269
19.4.1 一个if语句需要一个测试……269
19.4.2 get_user方法……272
19.4.3 在登录视图中使用自定义的验证后端……273
19.5 使用驭件的另一个原因：减少重复……274
19.5.1 使用驭件的返回值……277
19.5.2 在类一级上打补丁……278
19.6 关键时刻：功能测试能通过吗……279
19.7 理论上正常，那么实际呢……281
19.8 完善功能测试，测试退出功能……283
第 20 章 测试固件和一个显式等待装饰器……285
20.1 事先创建好会话，跳过登录过程……285
20.2 显式等待辅助方法最终版：wait装饰器……290
第 21 章 服务器端调试技术……293
21.1 实践是检验真理的唯一标准：在过渡服务器中捕获最后的问题……293
21.2 在服务器上通过环境变量设定机密信息……295
21.3 调整功能测试，以便通过POP3测试真实的电子邮件……296
21.4 在过渡服务器中管理测试数据库……299
21.4.1 创建会话的Django管理命令……300
21.4.2 让功能测试在服务器上运行管理命令……301
21.4.3 直接在Python代码中使用Fabric……302
21.4.4 回顾：在本地服务器和过渡服务器中创建会话的方式……303
21.5 集成日志相关的代码……304
21.6 小结……305

第 22 章　完成“My Lists”页面：由外而内的 TDD ……306
22.1　对立技术：“由内而外” ……306
22.2　为什么选择使用“由外而内” ……307
22.3　“My Lists”页面的功能测试……307
22.4　外层：表现层和模板……309
22.5　下移一层到视图函数（控制器） ……309
22.6　使用由外而内技术，再让一个测试通过……310
22.6.1　快速重组模板的继承层级……311
22.6.2　使用模板设计API……311
22.6.3　移到下一层：视图向模板中传入什么……313
22.7　视图层的下一个需求：新建清单时应该记录属主……313
22.8　下移到模型层……315
第 23 章　测试隔离和“倾听测试的心声”……319
23.1　重温抉择时刻：视图层依赖于尚未编写的模型代码……319
23.2　首先尝试使用驭件实现隔离……320
23.3　倾听测试的心声：丑陋的测试表明需要重构 ……323
23.4　以完全隔离的方式重写视图测试……323
23.4.1　为了新测试的健全性，保留之前的整合测试组件……324
23.4.2　完全隔离的新测试组件……324
23.4.3　站在协作者的角度思考问题……324
23.5　下移到表单层……329
23.6　下移到模型层……332
23.7　关键时刻，以及使用模拟技术的风险……335
23.8　把层与层之间的交互当作“合约” ……336
23.8.1　找出隐形合约……337
23.8.2　修正由于疏忽导致的问题……338
23.9　还缺一个测试……339
23.10　清理：保留哪些整合测试……340
23.10.1　删除表单层多余的代码……340
23.10.2　删除以前实现的视图……341
23.10.3　删除视图层多余的代码……342
23.11　总结：什么时候编写隔离测试，什么时候编写整合测试 ……343
23.11.1　以复杂度为准则……344
23.11.2　两种测试都要写吗……344
23.11.3　继续前行……344
第 24 章　持续集成……346
24.1　安装Jenkins……346
24.2　配置Jenkins……347

24.2.1 首次解锁 348
24.2.2 现在建议安装的插件 348
24.2.3 配置管理员用户 348
24.2.4 添加插件 350
24.2.5 告诉Jenkins到哪里寻找Python 3和Xvfb 350
24.2.6 设置HTTPS 351
24.3 设置项目 351
24.4 第一次构建 352
24.5 设置虚拟显示器，让功能测试能在无界面的环境中运行 354
24.6 截图 356
24.7 如有疑问，增加超时试试 359
24.8 使用PhantomJS运行QUnit JavaScript测试 359
24.8.1 安装node 359
24.8.2 在Jenkins中添加构建步骤 361
24.9 CI服务器能完成的其他操作 362

第 25 章 简单的社会化功能、页面模式以及练习 363

25.1 有多个用户以及使用addCleanup的功能测试 363
25.2 页面模式 365
25.3 扩展功能测试测试第二个用户和“My Lists”页面 367
25.4 留给读者的练习 368

第 26 章 测试运行速度的快慢和炽热的岩浆 371

26.1 正题：单元测试除了运行速度超快之外还有其他优势 372
26.1.1 测试运行得越快，开发速度越快 372
26.1.2 神赐的心流状态 372
26.1.3 经常不想运行速度慢的测试，导致代码变坏 373
26.1.4 现在还行，不过随着时间推移，整合测试会变得越来越慢 373
26.1.5 别只听我一个人说 373
26.1.6 单元测试能驱使我们实现好的设计 373
26.2 纯粹的单元测试有什么问题 373
26.2.1 隔离的测试难读也难写 373
26.2.2 隔离测试不会自动测试集成情况 374
26.2.3 单元测试几乎不能捕获意料之外的问题 374
26.2.4 使用驭件的测试可能和实现方式联系紧密 374
26.2.5 这些问题都可以解决 374
26.3 合题：我们到底想从测试中得到什么 374
26.3.1 正确性 374
26.3.2 简洁可维护的代码 375
26.3.3 高效的工作流程 375
26.3.4 根据所需的优势评估测试 375

26.4 架构方案······375
26.4.1 端口和适配器（或六边形、简洁）架构······376
26.4.2 函数式核心，命令式外壳······377
26.5 小结······377
遵从测试山羊的教诲······379
附录 A PythonAnywhere······381
附录 B 基于类的 Django 视图······385
附录 C 使用 Ansible 配置服务器······394
附录 D 测试数据库迁移······398
附录 E 行为驱动开发······403
附录 F 构建一个 REST API：JSON、Ajax 和 JavaScript 模拟技术······416
附录 G Django-Rest-Framework······433
附录 H 速查表······443
附录 I 接下来做什么······447
附录 J 示例源码······451
参考书目······453
作者简介······454
封面介绍······454

前言

我试图通过这本书与世人分享我从黑客变成软件工程师的过程。本书主要介绍测试，但很快你就会发现，除此之外还有很多其他内容。

感谢你阅读本书。

如果你购买了本书，我十分感激。如果你看的是免费在线版，我仍然要感谢你，因为你确定这本书值得花时间来阅读。谁知道呢，说不定等你读完之后，会决定为自己或朋友买一本纸质书。

如果你有任何评论、疑问或建议，希望你能写信告诉我。你可以通过电子邮件直接和我联系，地址是 obeythetestinggoat@gmail.com；或者在 Twitter 上联系我，我的用户名是 @hjwp。你还可以访问本书的网站和博客（http://www.obeythetestinggoat.com/），以及邮件列表（https://groups.google.com/forum/#!forum/obey-the-testing-goat-book）。

希望阅读本书能让你身心愉悦，就像我在写作本书时感到享受一样。

为什么要写一本关于测试驱动开发的书

我知道你会问："你是谁，为什么要写这本书，我为什么要读这本书？"

我至今仍然处在编程事业的初期。人们说，不管从事什么工作，都要历经从新手到熟手的过程，最终有可能成为大师。我要说的是，我最多算是个熟练的程序员。但我很幸运，在事业的早期阶段就结识了一群测试驱动开发（Test-Driven Development，TDD）的狂热爱好者，这对我的编程事业产生了极大影响，让我迫不及待地想和所有人分享这段经历。可以说，我很积极地做出了最近这次转变，而且这段学习经历现在还历历在目，我希望能让初学者感同身受。

我在开始学习 Python 时（看的是 Mark Pilgrim 写的 *Dive Into Python*），偶然知道了 TDD 的概念。我当时就认为："是的，我绝对知道这个概念的意义所在。"或许你第一次听说 TDD 时也有类似的反应吧。它听起来像是一个非常合理的方案，一个需要养成的非常好的习惯——就像经常刷牙。

随后我做了第一个大型项目。你可能猜到了，有项目就会有客户，有最后期限，有很多事情要做。于是，所有关于 TDD 的好想法都被抛诸脑后。

的确，这对项目没什么影响，对我也没影响。

但只是在初期如此。

一开始，我知道并不真的需要使用 TDD，因为我做的是个小网站，手动检查就能轻易测试出是否能用。在这儿点击链接，在那儿选中下拉菜单选项，就应该有预期的效果，很简单。编写整套测试程序听起来似乎要花费很长时间，而且经过整整三周成熟的代码编写经历，我自负地认为自己已经成为一名出色的程序员了，我能顺利完成这个项目，这没什么难度。

随后，项目变得复杂得可怕，这很快暴露了我的经验不足。

项目不断变大。系统的不同部分之间要开始相互依赖。我尽量遵守良好的开发原则，例如“不要自我重复”（Don’t Repeat Yourself，DRY），却被带进了一片危险地带。我很快就用到了多重继承，类的继承有八个层级深，还用到了 `eval` 语句。

我不敢修改代码，不再像以前一样知道什么依赖什么，也不知道修改某处的代码可能会导致什么后果。噢，天呐，我觉得那部分继承自这里，不，不是继承，是重新定义了，可是却依赖那个类变量。嗯，好吧，如果我再次重定义以前重定义的部分，应该就可以了。我会检查的，可是检查变得更难了。网站中的内容越来越多，手动点击变得不切实际了。最好别动这些能运行的代码，不要重构，就这么凑合吧。

很快，代码就变得像一团麻，丑陋不堪。开发新功能变得很痛苦。

在此之后不久，我幸运地在 Resolver Systems 公司（现在叫 PythonAnywhere）找到了一份工作。这个公司遵循极限编程（Extreme Programming，XP）开发理念。他们向我介绍了严密的 TDD。

虽然之前的经验的确让我认识到自动化测试的好处，但我在每个阶段都心存疑虑。“我的意思是，测试通常来说可能是个不错的主意，但果真如此吗？全部都要测试吗？有些测试看起来完全是在浪费时间……什么？除了单元测试之外还要做功能测试？得了吧，这是多此一举！还要走一遍测试驱动开发中的‘测试 / 小幅度代码改动 / 测试’循环？太荒谬了！我们不需要这种婴儿学步般的过程！既然我们知道正确的答案是什么，为什么不直接跳到最后一步呢？”

相信我！我审视过每一条规则，给每一条捷径提出过建议，为 TDD 的每一个看似毫无意义的做法寻找过理由，最终，我发现了采用 TDD 的明智之处。我记不清在心里说过多少次“谢谢你，测试”，因为功能测试能揭示我们可能永远都无法预测的回归，单元测试能让我避免犯很愚蠢的逻辑错误。从心理学上讲，TDD 大大降低了开发过程中的压力，而且写出的代码让人赏心悦目。

那么，让我告诉你关于 TDD 的一切吧！

写作本书的目的

我写这本书的主要目的是要传授一种用于 Web 开发的方法，它可以让 Web 应用变得更好，

也能让开发者更愉快。一本书如果只包含一些上网搜索就能找到的知识，那它就没多大的意思了，所以本书不是 Python 句法指南，也不是 Web 开发教程。我希望教会你的，是如何使用 TDD 理念，更加稳妥地实现我们共同的神圣目标——**简洁可用的代码**。

即便如此，我仍会从零开始使用 Django、Selenium、jQuery 和 Mock 等工具开发一个 Web 应用，不断提到一个真实可用的示例。阅读本书之前，你无须了解这些工具。读完本书后，你会充分了解这些工具，并掌握 TDD 理念。

在极限编程实践中，我们总是结对编程。写这本书时，我设想自己和以前的自己结成对子，向以前的我解释如何使用这些工具，回答为什么要用这种特别的方式编写代码。所以，如果我表现得有点儿屈尊俯就，那是因为我不是那么聪明，我要对自己很有耐心。如果觉得我说话冒犯了你，那是因为我有点儿烦人，经常不认同别人的说法，所以有时要花很多时间论证，说服自己接受他人的观点。

本书结构

我将这本书分成了三个部分。

- *第一部分（第 1~7 章）：基础知识*

  开门见山，介绍如何使用 TDD 开发一个简单的 Web 应用。我们会先（用 Selenium）写一个功能测试，然后介绍 Django 的基础知识，包括模型、视图和模板。在每个阶段，我们都会编写严格的单元测试。除此之外，我还会向你引荐测试山羊。

- *第二部分（第 8~17 章）：Web 开发要素*

  介绍 Web 开发过程中一些棘手但不可避免的问题，并展示如何通过测试解决这些问题，包括静态文件、部署到生产环境、表单数据验证、数据库迁移和令人畏惧的 JavaScript。

- *第三部分（第 18~26 章）：高级话题*

  介绍模拟技术、集成第三方系统、测试固件、由外而内的 TDD 流程以及持续集成（Continuous Integration，CI）。

排版约定

本书使用了下列排版约定。

- **黑体**

  表示新术语或强调的内容。

- 等宽字体（`Constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

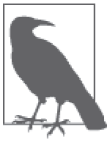
- 加粗等宽字体（**`Constant width bold`**）

  表示应该由用户输入的命令或其他文本。

  偶尔使用 `[...]` 符号表示省略了一些内容，截断较长的输出，或者跳到相关的内容。



该图标表示提示或建议。



该图标表示提示、建议或一般注记。



该图标表示警告或警示。

提交勘误

发现了错误或错别字？本书的相关资源放在 GitHub 上，欢迎你随时提交工单和拉取请求：https://github.com/hjwp/Book-TDD-Web-Dev-Python/。

如果发现中文版有错误或错别字，欢迎提交勘误至 http://www.ituring.com.cn/book/2052。

使用代码示例

代码示例可到 https://github.com/hjwp/book-example/ 下载，各章的代码都放在单独的分支中，请到 http://www.ituring.com.cn/book/2052“随书下载”处下载。附录 J 中有这个仓库的使用方法。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如：“*Test-Driven Development with Python*, *2nd edition*, by Harry Percival (O’Reilly). Copyright 2017 Harry Percival, 978-1-491-95870-4.”

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O’Reilly Safari

Safari（以前叫 Safari Books Online，http://www.safaribookson line.com）是会员制平台，为企业、政府、教学人员和个人提供培训和参考资料。

会员可以访问上千种图书、培训视频、学习路径、交互式教程和精心制定的播放列表。这些资源由 250 多家出版社提供，包括 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology，等等。

详情请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：
hhttp://shop.oreilly.com/product/0636920029533.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

电子书

如需购买本书电子版，请扫描以下二维码。

准备工作和应具备的知识

我假设读者具备了如下的知识，电脑中还应该安装一些软件。

了解Python 3，会编程

写这本书时，我考虑到了初学者。但如果你刚接触编程，我假设你已经学习了 Python 基础知识。如果还没学，请阅读一份 Python 初学者教程，或者买一本入门书，比如 *Dive Into Python* 或《“笨办法”学 Python》，或者出于兴趣，看一下《Python 游戏编程快速上手》。这三本都是很好的入门书。

如果你是经验丰富的程序员，但刚接触 Python，阅读本书应该没问题。Python 简单易懂。

本书中我用的是 Python 3。我在 2013—2014 年写这本书时，Python 3 已经发布好几年了，全世界的开发者正处在一个拐点上，他们更倾向于选择使用 Python 3。可以参照本书内容在 Mac、Windows 和 Linux 中实践。在各种操作系统中安装 Python 的详细说明后文会介绍。

> 本书内容在 Python 3.6 中测试过。如果你使用的是较低的版本，可能会发现细微的差别（比如 f 字符串句法），所以如果可以，最好升级 Python。

我不建议使用 Python 2，因为它和 Python 3 之间的区别太大。如果碰巧你的下一个项目使用的是 Python 2，仍然可以运用从本书中学到的知识。不过，当你得到的程序输出和本书不一样时，要花时间判断是因为用了 Python 2，还是因为你确实犯了错——这么做太浪费时间了。

如果你想使用 PythonAnywhere（这是我就职的 PaaS 创业公司），不愿在本地安装 Python，可以先快速阅读一遍附录 A。

无论如何，我希望你能使用 Python，知道如何从命令行启动 Python，也知道如何编辑和运行 Python 文件。再次提醒，如果你有任何疑问，看一下我前面推荐的三本书。

如果你已经安装了 Python 2，担心再安装 Python 3 会破坏之前的版本，那大可以放心，Python 3 和 Python 2 可以相安无事地共存于同一个系统中，使用虚拟环境的话（本书就是）更是如此。

HTML的工作方式

我还假定你基本了解 Web 的工作方式，知道什么是 HTML、什么是 POST 请求等。如果你对这些概念不熟悉，那么需要找一份 HTML 基础教程看一下，webplateform 网站上列出了一些。如果你知道如何在电脑中创建 HTML 页面，并在浏览器中查看，也知道表单以及它的工作方式，那么或许你就符合我的要求。

Django

本书使用 Django 框架，这（或许）是 Python 世界最为人认可的 Web 框架。本书不要求读者事先了解 Django，但如果你刚接触 Python、刚接触 Web 开发，也刚接触测试，偶尔会觉得话题太多，有太多的概念要理解。如果发生了这样的情况，我建议你先把本书放下，找份 Django 教程看看。DjangoGirls 是我所知的对新手最友好的教程。官方教程（https://docs.djangoproject.com/en/1.11/intro/tutorial01/）对有经验的程序员来说是不错的选择。

Django 的安装说明参见后文。

JavaScript

本书后半部分有少量 JavaScript。如果你不了解 JavaScript，先别担心。如果你觉得有些看不懂，我到时会推荐一些参考资料给你。

关于 IDE

如果你来自 Java 或 .NET 领域，可能非常想使用 IDE（集成开发环境）编写 Python 代码。IDE 中有各种实用的工具，例如 VCS 集成。Python 领域也有一些很棒的 IDE。刚开始我也用过一个 IDE，它对我最初的几个项目很有用。

我能建议（只是建议）你**别用** IDE 吗？至少在阅读本书时别用。在 Python 领域，IDE 不是那么重要。写作本书时，我假定你只使用一个简单的文本编辑器和命令行。某些时候，它们是你所能使用的全部工具（例如在服务器中），所以刚开始时值得花时间学习使用基本的工具，理解它们是如何工作的。即便当你读完本书后决定继续使用 IDE 和其中的实用工具，这些基本的工具还是唾手可得。

需要安装的软件

除了 Python 之外，还要安装以下软件。

- Firefox Web 浏览器

  Selenium 其实能驱动任意一款主流浏览器，不过以 Firefox 举例最简单，因为它跨平台。而且使用 Firefox 还有另外一个好处——和公司利益没有多少关联。

- Git 版本控制系统

  Git 可在任何一个平台上使用。Windows 安装环境带有 Bash 命令行，这是本书所需的。

- 装有 Python 3、Django 1.11 和 Selenium 3 的虚拟环境

  Python 3.4+ 现在自带 virtualenv 和 pip（早期版本没有，这是一大进步）。搭建虚拟环境的详细说明参见后文。

- Geckodriver

  这是通过 Selenium 远程控制 Firefox 的驱动。在“安装 Firefox 和 Geckodriver”一节会给出下载链接。

针对 Windows 的说明

Windows 用户有时会觉得被开源世界忽略了，因为 macOS 和 Linux 太普遍了，很容易让人忘记在 Unix 之外还有一个世界。使用反斜线作为目录分隔符？盘符？这些是什么？不过，阅读本书时仍然可以在 Windows 中实践。下面是一些小提示。

1. 在 Windows 中安装 Git 时，一定要选择“Run Git and included Unix tools from the Windows command prompt”（在 Windows 命令提示符中运行 Git 和所含的 Unix 工具）。选择这个选项之后就能使用 Git Bash 了。把 Git Bash 作为主要命令提示符，你就能使用所有实用的 GNU 命令行工具，例如 `ls`、`touch` 和 `grep`，而且目录分隔符也使用斜线表示。

2. 在 Git 安装程序中，还要勾选“Use Windows’default console”（使用 Windows 的默认控制台），否则 Python 在 Git Bash 窗口中无法正常使用。

3. 安装 Python 3 时，除非已经安装了 Python 2 且想继续将它用作默认版本，否则一定要选中“Add Python 3.6 to PATH”（把 Python 3.6 添加到系统路径中，如图 P-1 所示），这样才能在命令行中顺利运行 Python。

> 测试所有软件是否正确安装的方法是，打开 Git Bash 命令提示符，在任意一个文件夹中执行命令 `python` 或 `pip`。



图 P-1：从安装程序将 Python 加入系统路径

针对 MacOS 的说明

MacOS 比 Windows 稍微正常一点儿，不过在 Python 3.4 之前，安装 `pip` 还是一项极具挑战性的任务。Python 3.4 发布后，安装方法变得简单明了。

- 使用下载的安装程序就能安装 Python 3.6，省去了很多麻烦。而且，这个安装程序也会自动安装 `pip`。
- Git 安装程序也能顺利运行。

测试这些软件是否正常安装的方法和 Windows 类似：打开一个终端，然后在任意位置执行命令 `git`、`python3` 或 `pip`。如果遇到问题，搜索关键字“system path”和“command not found”，就能找到解决问题的合适资源。

> 或许你还应该检验一下 Homebrew。在 Mac 安装众多 Unix 工具的方式中，它是唯一可信赖的。[1] 虽然现在 Python 安装程序做得不错，但将来你可能会用到 Homebrew。要使用 Homebrew，需下载大小为 1.1 GB 的 Xcode。不过这有个好处——你得到了一个 C 编译器。

Git默认使用的编辑器和其他基本配置

后文我会逐步介绍如何使用 Git，不过现在最好做些配置。例如，首次提交时，默认情况下会弹出 vi，这可能让你手足无措。鉴于 vi 有两种模式，因此你有两个选择。其一，学一

注 1：不过我不建议使用 Homebrew 安装 Firefox，因为 Homebrew 会把 Firefox 二进制文件放到一个陌生的位置，Selenium 找不到。虽然这个问题可以解决，但是以常规的方式安装更简单。

些基本的 vi 命令（**按 i 键进入插入模式，输入文本后再按 `<Esc>` 键返回常规模式，然后输入 `:wq<Enter>` 写入文件并退出**）。学会这些命令后，你就加入了一个互助会，这里的人们知道怎么使用这个古老而让人崇敬的文本编辑器。

另外一个选择是直接拒绝这种穿越到 20 世纪 70 年代的荒唐行为，而是配置 Git，让它使用你选择的编辑器。按 `<Esc>` 键，再输入 `:q!`，退出 vi，然后修改 Git 使用的默认编辑器。具体方法参见介绍 Git 基本配置的文档。

安装Firefox和Geckodriver

从 https://www.mozilla.org/firefox/ 可下载 Windows 和 macOS 的 Firefox 安装包。Linux 可能已经预装了 Firefox；如果没有，使用包管理器安装。

Geckodriver 可从 https://github.com/mozilla/geckodriver/releases 下载。下载后解压，放到系统路径中的某个位置。

- 对 macOS 或 Linux 来说，可以放在 ~/.local/bin 目录中。
- 对 Windows 来说，可以放在 Python 的 Scripts 文件夹中。

为了确认是否成功安装，打开一个 Bash 控制台，执行下述命令：

```
geckodriver --version
geckodriver 0.17.0

The source code of this program is available at
https://github.com/mozilla/geckodriver.

This program is subject to the terms of the Mozilla Public License 2.0.
You can obtain a copy of the license at https://mozilla.org/MPL/2.0/.
```

如果无法执行这个命令，可能是因为 ~/.local/bin 不在 `PATH` 中（针对 Mac 和 Linux 系统）。这个文件夹最好加到 `PATH` 中，因为使用 `pip install --user` 安装的 Python 包都存储在这里。把这个文件夹添加到 .bashrc 文件中的方法如下所示[2]：

```
echo 'PATH=~/.local/bin:$PATH' >> ~/.bashrc
```

然后关闭终端，重新打开，再次确认能否执行 `geckodriver --version` 命令。

搭建虚拟环境

Python 项目所需的环境使用 `virtualenv`（virtual environment 的简称）搭建。在不同项目的虚拟环境中可以使用不同的包（例如不同版本的 Django，甚至是不同版本的 Python）。而且虚拟环境中的包不是全局安装的，因此无须 root 权限。

Python 从 3.4 版开始集成了用于搭建虚拟环境的 `virtualenv` 工具，不过我始终建议使用 `virtualenvwrapper` 这个辅助工具。先安装 `virtualenvwrapper`（对 Python 版本没有要求）：

注 2：.bashrc 是 Bash 的初始化文件，在家目录中。每次运行 Bash 都会运行这个文件。

```
# 在Windows中
pip install virtualenvwrapper
# 在macOS/Linux中
pip install --user virtualenvwrapper
# 然后让Bash自动加载virtualenvwrapper
echo "source virtualenvwrapper.sh" >> ~/.bashrc
source ~/.bashrc
```

在 Windows 中，`virtualenvwrapper` 只能在 Git Bash 中使用，而不能在常规的命令行中使用。

`virtualenvwrapper` 把所有虚拟环境都放在一个地方，而且为激活和停用虚拟环境提供了便利的工具。

下面创建一个名为“superlists”[3] 的虚拟环境，并在里面安装 Python 3：

```
# 在macOS/Linux中
mkvirtualenv --python=python3.6 superlists
# 在Windows中
mkvirtualenv --python=`py -3.6 -c"import sys; print(sys.executable)"` superlists
#（为了得到一个装有Python 3.6的虚拟环境，我们绕了点弯子）
```

激活和停用虚拟环境

阅读本书时，一定要先“激活”你的虚拟环境。你之所以能看出我们处在虚拟环境中，通常是因为提示符中有 `(superlists)`，例如：

```
$
(superlists) $
```

创建虚拟环境之后，就直接激活了虚拟环境。你可以执行 `which python` 命令再次确认：

```
(superlists) $ which python
/home/harry/.virtualenvs/superlists/bin/python
#（在Windows中会显示为下面这样
# /C/Users/IEUser/.virtualenvs/superlists/Scripts/python）

(superlists) $ deactivate
$ which python
/usr/bin/python
$ python --version
Python 2.7.12  # 在我的设备中，虚拟环境外部的“python”默认为Python 2

$ workon superlists
(superlists) $ which python
/home/harry/.virtualenvs/superlists/bin/python
(superlists) $ python --version
Python 3.6.0
```

注 3：你可能会问为什么叫“superlists”？我可不想剧透！下一章你就知道了。

激活虚拟环境的命令是 workon superlists。若想确认有没有激活，可以看命令提示符中有没有 (superlists) $，或者执行 which python 命令。

安装Django和Selenium

我们将安装 Django 1.11 和最新版 Selenium，即 Selenium 3：

```
(superlists) $ pip install "django<1.12" "selenium<4"
Collecting django==1.11.3
  Using cached Django-1.11.3-py2.py3-none-any.whl
Collecting selenium<4
  Using cached selenium-3.4.3-py2.py3-none-any.whl
Installing collected packages: django, selenium
Successfully installed django-1.11.3 selenium-3.4.3
```

无法激活虚拟环境时可能会看到的一些错误消息

对刚接触虚拟环境的人来说，肯定会经常忘记激活虚拟环境（说实话，**老手**也经常犯这个错，比如我）。这时，你会看到一个错误消息，其中的重要部分如下所示：

```
ImportError: No module named selenium
```

或者是：

```
ImportError: No module named django.core.management
```

如果遇到这种错误，不要慌，先看看命令提示符中有没有 (superlists)。通常只需执行 workon superlists 就能解决问题。

除此之外，可能还会遇到这个错误：

```
bash: workon: command not found
```

这表明你前面少做了一步，没有把 virtualenvwrapper 添加到 .bashrc 中。从前文中找到 echo source virtualenvwrapper.sh 命令，再执行一遍。

```
'workon' is not recognized as an internal or external command,
operable program or batch file.
```

这表明你打开的是 Windows 的默认命令提示符 cmd，而不是 Git Bash。关掉 cmd，打开 Git Bash。

编程快乐！

上述说明对你没什么用，或者你有更好的说明？请给我发电子邮件吧，地址是 obeythetestinggoat@gmail.com。

配套视频

我为本书录制了一套十集的配套视频（http://oreil.ly/1svTFqB）[1]，主要针对第一部分的内容。如果你更适合通过视频学习，建议你看看。除了书中的内容之外，这套视频还能让你直观地感受 TDD 流程，了解如何在测试和代码之间切换，与此同时保持思路清晰。

我还特意穿了一件亮黄色 T 恤。



注 1：这套视频没有针对第 2 版更新，不过内容基本上依然适用。

致谢

要感谢的人很多，没有他们就不会有这本书，就算有也会写得比现在差。

首先感谢某出版社的 Greg，他是第一个鼓励我、让我相信我能写完这本书的人。虽然你们公司在版权问题上思想过于退化，但我还是要感激你个人对我的信任。

感谢 Michael Foord，他以前也是 Resolver Systems 的员工。我写书的最初灵感便来自于他，因为他自己也写了一本书。感谢他一直支持这个项目。感谢我的老板 Giles Thomas，他傻傻地允许他的另一位员工也去写书（不过，我觉得他现在已经修改了标准的雇佣合同，加上了“禁止写书”的条款）。同样也感谢你不断增长的智慧，把我带入了测试领域。

感谢我的另外两位同事，Glenn Jones 和 Hansel Dunlop。你们总是给我提供非常宝贵的意见，而且在过去一年中耐心地陪我录制单轨对话。

感谢我的妻子 Clementine 和家人，没有他们的支持和耐心，我绝对无法写完这本书。很多时间本该和家人在一起难忘地度过，我却把它们都花在了写作上，为此我感到抱歉。开始写作时，我不知道这本书会对我的生活产生什么影响（“在闲暇时间写怎么样？听起来可行……”）。没有你们的支持，我写不完这本书。

感谢技术审校人员 Jonathan Hartley、Nicholas Tollervey 和 Emily Bache。感谢你们的鼓励和珍贵的反馈。尤其是 Emily，她认真阅读了每一章。感谢 Nick 和 Jon，由衷感谢。感谢你们在我身旁，让写作的过程变得不那么孤单。没有你们，这本书的内容会像傻瓜一样废话连篇。

感谢每个放弃自己的时间把意见反馈给我的人，感谢你们的热心肠：Gary Bernhardt、Mark Lavin、Matt O’Donnell、Michael Foord、Hynek Schlawack、Russell Keith-Magee、Andrew Godwin 和 Kenneth Reitz。你们比我聪慧得多，让我避免说一些愚蠢的事情。当然，书中还有很多愚蠢的内容，不过责任肯定不在你们。

感谢我的编辑 Meghan Blanchette，她是一位非常友善可爱的监工。谢谢你为我规划时间，制止我愚蠢的想法，让本书的写作在正确的轨道上行进。感谢 O’Reilly 出版社其他所有为我提供帮助的人，包括 Sarah Schneider、Kara Ebrahim 和 Dan Fauxsmith，感谢你们让我继续使用英式英语。感谢 Charles Roumeliotis 在行文风格和语法上给我的帮助。虽然我们对

芝加哥学派引用和标点符号规则的优势有不同观点，但有你在身边我仍然高兴。感谢设计部门为封面绘制了一头山羊。

特别感谢预览版的读者，感谢你们挑出拼写错误，给我反馈和建议，感谢你们提出各种有助于使本书学习曲线变平滑的方法，感谢你们中的大多数人给我鼓励和支持，让我一直写下去。感谢 Jason Wirth、Dave Pawson、Jeff Orr、Kevin De Baere、crainbf、dsisson、Galeran、Michael Allan、James O'Donnell、Marek Turnovec、Sooner Bourne、julz、Cody Farmer、William Vincent、Trey Hunner、David Souther、Tom Perkin、Sorcha Bowler、Jon Poler、Charles Quast、Siddhartha Naithani、Steve Young、Roger Camargo、Wesley Hansen、Johansen Christian Vermeer、Ian Laurain、Sean Robertson、Hari Jayaram、Bayard Randel、Konrad Korzel、Matthew Waller、Julian Harley、Barry McClendon、Simon Jakobi、Angelo Cordon、Jyrki Kajala、Manish Jain、Mahadevan Sreenivasan、Konrad Korzel、Deric Crago、Cosmo Smith、Markus Kemmerling、Andrea Costantini、Daniel Patrick、Ryan Allen、Jason Selby、Greg Vaughan、Jonathan Sundqvist、Richard Bailey、Diane Soini、Dale Stewart、Mark Keaton、Johan Wärlander、Simon Scarfe、Eric Grannan、Marc-Anthony Taylor、Maria McKinley、John McKenna、Rafał Szymański、Roel van der Goot、Ignacio Reguero、TJ Tolton、Jonathan Means、Theodor Nolte、Jungsoo Moon、Craig Cook、Gabriel Ewilazarus、Vincenzo Pandolfo、David "farbish2"、Nico Coetzee、Daniel Gonzalez、Jared Contrascere、赵亮等很多人。如果我遗漏了你的名字，你绝对有权感到委屈。我当然非常感激你，请给我写封信，我会尽我所能把你的名字加上。

最后，我要感谢你，现在的读者，感谢你决定阅读这本书，希望你喜欢。

第2版附加致谢

感谢第 2 版的编辑 Nan Barber，感谢 Susan Conant、Kristen Brown 和 O'Reilly 整个团队。再次感谢 Emily 和 Jonathan，感谢你们的技术审阅，还要感谢 Edward Wong 细致的笔记。倘若书中还有错误和不足，责任都在我。

感谢免费版的读者们，感谢你们提出的意见和建议，有些读者甚至发起了拉取请求。感谢 Emre Gonulates、Jésus Gómez、Jordon Birk、James Evans、Iain Houston、Jason DeWitt、Ronnie Raney、Spencer Ogden、Suresh Nimbalkar、Darius、Caco、LeBodro、Jeff、wasabigeek、joegnis、Lars、Mustafa、Jared、Craig、Sorcha、TJ、Ignacio、Roel、Justyna、Nathan、Andrea、Alexandr、bilyanhadzhi、mosegontar、sfarzy、henziger、hunterji、das-g、juanriaza、GeoWill、Windsooon、gonulate，等等。我肯定遗漏了一些名字，为此我深感抱歉。

第一部分

TDD和Django基础

第一部分我要介绍**测试驱动开发**（Test-Driven Development，TDD）的基础知识。我们会从零开始开发一个真实的 Web 应用，而且每个阶段都要先写测试。

这一部分涵盖使用 Selenium 完成的功能测试以及单元测试，还会介绍二者之间的区别。我会介绍 TDD 流程，我称之为“单元测试 / 编写代码”循环。我们还要做些重构，说明怎么结合 TDD 使用。因为版本控制对重要的软件工程来说是基本需求，所以我们还会用到版本控制系统（Git）。我会介绍何时以及如何提交，如何把提交集成到 TDD 和 Web 开发的流程中。

我们要使用 Django，它（或许）是 Python 领域之中最受欢迎的 Web 框架。我会试着慢慢介绍 Django 的概念，一次一个，除此之外还会提供很多扩展阅读资料的链接。如果你完全是刚接触 Django，那么我极力推荐你花时间阅读这些资料。如果你感觉有点儿茫然，花几小时读一遍 Django 的官方教程，然后再回来阅读本书。

你还会结识测试山羊……

复制粘贴时要小心

如果你看的是电子版，那么在阅读的过程之中就会很自然地会想要复制粘贴书中的代码清单。如果不这么做的话效果会更好：动手输入能形成肌肉记忆，感觉也更真实。你偶尔会打错字，这是无法避免的，调试错误也是一项需要学习的重要技能。

除此之外，你还会发现 PDF 格式相当诡异，复制粘贴时经常会有意想不到的事情发生……

第 1 章

使用功能测试协助安装Django

TDD 不是天生就会的技术，而是像武术一样的一种技能。就像在功夫电影中一样，你需要一个脾气不好、不可理喻的师傅来强制你学习。我们的师傅是测试山羊。

1.1　遵从测试山羊的教诲，没有测试什么也别做

在 Python 测试社区中，测试山羊是 TDD 的非官方吉祥物。测试山羊对不同的人有不同的意义。对我来说，它是我脑海中的一个声音，告诉我要一直走在测试这条正确的道路上，就像卡通片中浮现在肩膀上的天使或魔鬼一样，只是没那么咄咄逼人。我希望借由这本书，让测试山羊也扎根于你的脑海中。

虽然还不太确定要做什么，但我们已经决定要开发一个网站。Web 开发的第一步通常是安装和配置 Web 框架。**下载这个，安装那个，配置那个，运行这个脚本……**但是，使用 TDD 时要转换思维方式。做测试驱动开发时，你的心里要一直记着测试山羊，像山羊一样专注，咩咩地叫着："先测试，先测试！"

在 TDD 的过程中，第一步始终一样：**编写测试**。

首先要编写测试，**然后**运行，看是否和预期一样失败，**只有**失败了才能继续下一步——编写应用程序。请模仿山羊的声调复述这个过程。我就是这么做的。

山羊的另一个特点是一次只迈一步。因此，不管山壁多么陡峭，它们都不会跌落。看看图 1-1 里的这只山羊！



图 1-1：山羊比你想象的要机敏（来源：Flickr 用户 Caitlin Stewart）

我们会碎步向前。使用流行的 Python Web 框架 Django 开发这个应用。

首先，要检查是否安装了 Django，并且能够正常运行。检查的方法是，在本地电脑中能否启动 Django 的开发服务器，并在浏览器中查看能否打开网页。使用浏览器自动化工具 Selenium 完成这个任务。

在你想保存项目代码的地方新建一个 Python 文件，命名为 functional_tests.py，并输入以下代码。如果你喜欢一边输入代码一边像山羊那样轻声念叨，或许会有所帮助：

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('http://localhost:8000')

assert 'Django' in browser.title
```

这是我们编写的第一个**功能测试**（Functional Test，FT）。后面我会深入说明什么是功能测试，以及它和单元测试的区别。现在，只要能理解这段代码做了什么就行。

- 启动一个 Selenium webdriver，打开一个真正的 Firefox 浏览器窗口。
- 在这个浏览器中打开我们期望本地电脑伺服的网页。
- 检查（做一个测试断言）这个网页的标题中是否包含单词“Django”。

我们尝试运行一下：

```
$ python functional_tests.py
  File ".../selenium/webdriver/remote/webdriver.py", line 268, in get
    self.execute(Command.GET, {'url': url})
  File ".../selenium/webdriver/remote/webdriver.py", line 256, in execute
    self.error_handler.check_response(response)
  File ".../selenium/webdriver/remote/errorhandler.py", line 194, in
check_response
    raise exception_class(message, screen, stacktrace)
selenium.common.exceptions.WebDriverException: Message: Reached error page: abo
ut:neterror?e=connectionFailure&u=http%3A//localhost%3A8000/[...]
```

你应该会看到弹出了一个浏览器窗口，尝试打开 localhost:8000，然后显示“无法连接”错误页面。这时回到终端，你会看到一个显眼的错误消息，说 Selenium 遇到了一个错误页面。接着，你会看到 Firefox 窗口停留在桌面上，等待你关闭。这可能会让你生气，我们稍后会修正这个问题。

> 如果看到关于导入 Selenium 的错误，或者让你查找“geckodriver”错误，或许你应该往前翻，看一下“准备工作和应具备的知识”。

现在，得到了一个**失败测试**。这意味着，我们可以开始开发应用了。

> **别了，罗马数字**
>
> 很多介绍 TDD 的文章都喜欢以罗马数字为例，闹了笑话，甚至我一开始也是这么写的。如果你好奇，可以查看我在 GitHub 的页面，地址是 https://github.com/hjwp/。
>
> 以罗马数字为例有好也有坏。把问题简化，合理地限制在某一范围内，让你能很好地解说 TDD。
>
> 但问题是不切实际。因此我才决定要从零开始开发一个真实的 Web 应用，以此为例介绍 TDD。这是一个简单的 Web 应用，我希望你能把从中学到的知识运用到下一个真实的项目中。

1.2　让Django运行起来

你肯定已经读过“准备工作和应具备的知识”了，也安装了 Django。使用 Django 的第一步是创建**项目**，我们的网站就放在这个项目中。Django 为此提供了一个命令行工具：

```
$ django-admin.py startproject superlists
```

这个命令会创建一个名为 superlists 的文件夹，并在其中创建一些文件和子文件夹：

```
.
├── functional_tests.py
├── geckodriver.log
└── superlists
    ├── manage.py
    └── superlists
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```

在 superlists 文件夹中还有一个名为 superlists 的文件夹。这有点让人困惑，不过确实需要如此。回顾 Django 的历史，你会找到出现这种结构的原因。现在，重要的是知道 superlists/superlists 文件夹的作用是保存应用于整个项目的文件，例如 settings.py 的作用是存储网站的全局配置信息。

你还会注意到 manage.py。这个文件是 Django 的瑞士军刀，作用之一是运行开发服务器。我们来试一下。执行命令 **`cd superlists`**，进入顶层文件夹 superlists（我们会经常在这个文件夹中工作），然后执行：

```
$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

Django version 1.11.3, using settings 'superlists.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

暂时先不管关于“未应用迁移”的消息，第 5 章将讨论迁移。

这样，Django 的开发服务器便在设备中运行起来了。让这个命令一直运行着，再打开一个命令行窗口（进入刚刚打开的文件夹），在其中再次运行测试：

```
$ python functional_tests.py
$
```

因为打开了新的终端窗口，所以要先执行 `workon superlists` 命令激活虚拟环境。

我们在命令行中没执行多少操作，但你应该注意两件事：第一，没有丑陋的 `AssertionError` 了；第二，Selenium 弹出的 Firefox 窗口中显示的页面不一样了。

这虽然看起来没什么大不了，但毕竟是我们第一个通过的测试啊！值得庆祝。

如果感觉有点神奇，不太现实，为什么不手动查看开发服务器，打开浏览器访问 http://localhost:8000 呢？你会看到如图 1-2 所示的页面。

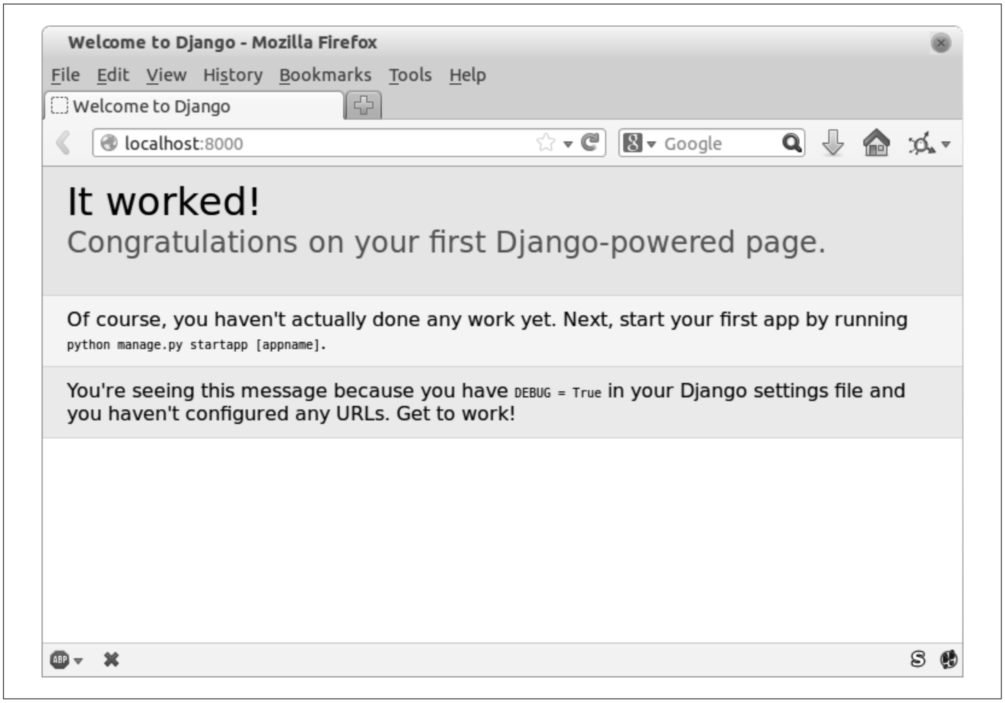


图 1-2：Django 可用了

如果想退出开发服务器，可以回到第一个 shell 中，按 Ctrl-C 键。

1.3 创建Git仓库

结束这章之前，还要做一件事：把作品提交到**版本控制系统**（Version Control System，VCS）。如果你是一名经验丰富的程序员，就无须再听我宣讲版本控制了。如果你刚接触 VCS，请相信我，它是必备工具。当项目在几周内无法完成，代码越来越多时，你需要一个工具查看旧版代码、撤销改动、放心地试验新想法，或者只是做个备份。测试驱动开发和版本控制关系紧密，所以我一定要告诉你如何在开发流程中使用版本控制系统。

好的，来做第一次提交。如果现在提交已经晚了，我表示歉意。我们使用 Git 作为 VCS，因为它是最棒的。

我们先把 functional_tests.py 移到 superlists 文件夹中。然后执行 `git init` 命令，创建仓库：

```
$ ls
superlists  functional_tests.py  geckodriver.log
$ mv functional_tests.py superlists/
$ cd superlists
$ git init .
Initialised empty Git repository in /.../superlists/.git/
```

自此工作目录都是顶层 superlists 文件夹

从现在起，我们会把顶层文件夹 superlists 作为工作目录。

（简单起见，我在命令列表中都将使用 /.../superlists/ 表示这个目录。但实际上，这个目录的真实地址可能是 /home/kind-reader-username/my-python-projects/superlists/。）

我提供的输入命令都假定在这个目录中执行。同样，如果我提到一个文件的路径，也是相对于这个顶层目录而言。因此，superlists/settings.py 是指次级文件夹 superlists 中的 settings.py。

理解了吗？如果有疑问，就查找 manage.py，你要和这个文件在同一个目录中。

现在，看一下要提交的文件：

```
$ ls
db.sqlite3  manage.py   superlists  functional_tests.py
```

db.sqlite3 是数据库文件，无须纳入版本控制。前面见过的 geckodriver.log 是 Selenium 的日志文件，也无须跟踪变化。我们要把这两个文件添加到一个特殊的文件 .gitignore 中，让 Git 忽略它们：

```
$ echo "db.sqlite3" >> .gitignore
$ echo "geckodriver.log" >> .gitignore
```

接下来，我们可以添加当前文件夹（“.”）中的其他内容了：

```
$ git add .
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   functional_tests.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/__pycache__/__init__.cpython-36.pyc
        new file:   superlists/__pycache__/settings.cpython-36.pyc
        new file:   superlists/__pycache__/urls.cpython-36.pyc
        new file:   superlists/__pycache__/wsgi.cpython-36.pyc
        new file:   superlists/settings.py
```

```
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py
```

糟糕，添加了很多 .pyc 文件，这些文件没必要提交。将其从 Git 中删掉，并添加到 .gitignore 中：

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-36.pyc'
rm 'superlists/__pycache__/settings.cpython-36.pyc'
rm 'superlists/__pycache__/urls.cpython-36.pyc'
rm 'superlists/__pycache__/wsgi.cpython-36.pyc'
$ echo "__pycache__" >> .gitignore
$ echo "*.pyc" >> .gitignore
```

现在，来看一下进展到哪里了。（你会看到，我使用 `git status` 的次数太多了，所以经常会使用别名 `git st`。我不会告诉你怎么做，你要自己探索 Git 别名的秘密！）

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   functional_tests.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .gitignore
```

看起来不错，可以做第一次提交了：

```
$ git add .gitignore
$ git commit
```

输入 `git commit` 后，会弹出一个编辑器窗口，让你输入提交消息。我写的消息如图 1-3 所示。[1]

注 1：是不是 vi 弹出后你不知道该做什么？或者，你是不是看到了一个消息，内容是关于账户识别的，其中还显示了 `git config --global user.username`？再次看一下“准备工作和应具备的知识”，里面有一些简单说明。

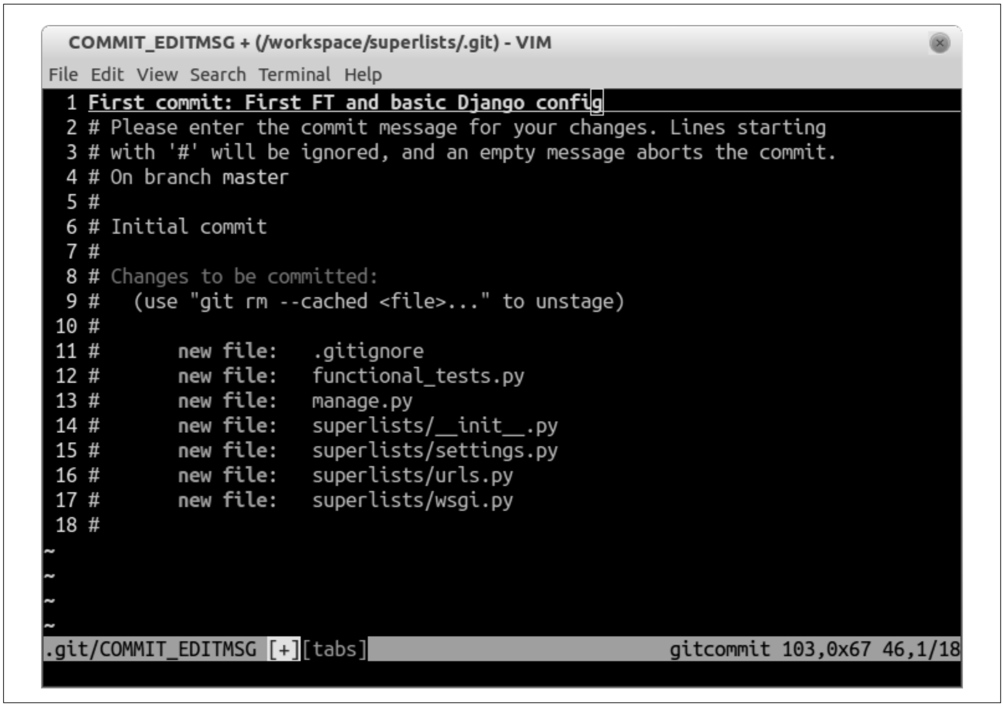


图 1-3：首次 Git 提交

> 如果你迫切想完成整个 Git 操作，此时还要学习如何把代码推送到云端的 VCS 托管服务中，例如 GitHub 或 BitBucket。如果你在阅读本书的过程中使用不同的电脑，会发现这么做很有用。具体的操作留给你去发掘，GitHub 和 BitBucket 的文档写得都很好。要不，你可以等到第 9 章，到时我们会使用其中一个服务做部署。

对 VCS 的介绍结束。祝贺你！你使用 Selenium 编写了一个功能测试，安装了 Django，并且使用 TDD 方式，以测试山羊赞许的、先写测试这种有保障的方式运行了 Django。在继续阅读第 2 章之前，先表扬一下自己吧，这是你应得的奖励。

第2章

使用unittest模块扩展功能测试

测试目前检查的是 Django 默认的“可用了”页面，修改一下，让其检查我们想在真实的网站首页中看到的内容。

是时候公布我们要开发什么类型的 Web 应用了——一个待办事项清单网站。开发这种网站说明我们始终追随时尚：很多年前所有的 Web 开发教程都介绍如何开发博客，后来一窝蜂地又介绍论坛和投票应用，现在时兴的是待办事项清单。

不过待办事项清单是个很好的例子。很明显，这种应用简单，只显示一个由文本字符串组成的列表，因此很容易得到一个最简可用的应用。而且可以使用各种方式扩展功能，例如使用不同的持久模型、添加最后期限、提醒和分享功能，还可以改进客户端 UI。另外，不必只局限于列出待办事项，这种应用可以列出任何事项。最重要的一点是，通过这种应用，可以演示 Web 开发过程中的所有主要步骤，以及如何在各个步骤中运用 TDD 理念。

2.1　使用功能测试驱动开发一个最简可用的应用

使用 Selenium 实现的测试可以驱动真正的网页浏览器，让我们能从用户的角度查看应用是如何**运作**的。因此，我们把这类测试叫作**功能测试**。

这意味着，功能测试在某种程度上可以作为应用的说明书。功能测试的作用是跟踪**用户故事**（User Story），模拟用户使用某个功能的过程，以及应用应该如何响应用户的操作。

> **术语：功能测试 = 验收测试 = 端到端测试**
>
> 我所说的功能测试，有些人喜欢称之为**验收测试**（acceptance test）或**端到端测试**（end-to-end test）。这类测试最重要的作用是从外部观察整个应用是如何运作的。另一个术语是**黑箱测试**（black box test），因为这种测试对所要测试的系统内部一无所知。

功能测试应该有一个人类可读、容易理解的故事。为了叙事清楚，可以把测试代码和代码注释结合起来使用。编写新功能测试时，可以先写注释，勾勒出用户故事的重点。这样写出的测试人类可读，甚至可以作为一种讨论应用需求和功能的方式分享给非程序员看。

TDD 常与敏捷软件开发方法结合在一起使用，我们经常提到的一个概念是“最简可用的应用”，即我们能开发出来的最简单的而且可以使用的应用。下面我们就来开发一个最简可用的应用，尽早试水。

最简可用的待办事项清单其实只要能让用户输入一些待办事项，并且用户下次访问应用时这些事项还在即可。

打开 functional_tests.py，编写一个类似下面的故事：

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()

# 伊迪丝听说有一个很酷的在线待办事项应用
# 她去看了这个应用的首页
browser.get('http://localhost:8000')

# 她注意到网页的标题和头部都包含“To-Do”这个词
assert 'To-Do' in browser.title

# 应用邀请她输入一个待办事项

# 她在一个文本框中输入了“Buy peacock feathers”（购买孔雀羽毛）
# 伊迪丝的爱好是使用假蝇做饵钓鱼

# 她按回车键后，页面更新了
# 待办事项表格中显示了“1: Buy peacock feathers”

# 页面中又显示了一个文本框，可以输入其他的待办事项
# 她输入了“Use peacock feathers to make a fly”（使用孔雀羽毛做假蝇）
# 伊迪丝做事很有条理

# 页面再次更新，她的清单中显示了这两个待办事项

# 伊迪丝想知道这个网站是否会记住她的清单
# 她看到网站为她生成了一个唯一的URL
# 而且页面中有一些文字解说这个功能

# 她访问那个URL，发现她的待办事项列表还在

# 她很满意，去睡觉了

browser.quit()
```

> **我们有个词来形容注释**
>
> 我开始在 Resolver 工作时，出于善意习惯在代码中加入密密麻麻的详细注释。我的同事告诉我：“哈利，我们有个词来形容注释，我们把注释叫作谎言。”我很诧异：可我在学校学到的是，注释是好的习惯啊？
>
> 他们说得夸张了。注释有其作用，可以添加上下文，说明代码的目的。他们的意思是，简单重复代码意图的注释毫无意义，例如：
>
> ```
> # 把wibble的值增加 1
> wibble += 1
> ```
>
> 这样的注释不仅毫无意义，还有一定危险，如果更新代码后没有修改注释，会误导别人。我们要努力做到让代码可读，使用有意义的变量名和函数名，保持代码结构清晰，这样就不再需要通过注释说明代码**做了什么**，只要偶尔写一些注释说明**为什么**这么做。
>
> 有些情况下注释很重要。后文会看到，Django 在其生成的文件中用到了很多注释，这是解说 API 的一种方式。而且还在功能测试中使用注释描述用户故事——把测试提炼成一个连贯的故事，确保我们始终从用户的角度测试。
>
> 这个领域中还有很多有趣的知识，例如**行为驱动开发**（Behaviour Driven Development，详情参见附录 E）和测试 DSL（Domain Specific Language，领域特定语言）。这些知识已经超出本书的范畴了。

你可能已经发现了，除了在测试中加入注释之外，我还修改了 `assert` 这行代码，让其查找单词“To-Do”，而不是“Django”。这意味着现在我们期望测试失败。运行这个测试。

首先，启动服务器：

```
$ python manage.py runserver
```

然后在另一个 shell 中运行测试：

```
$ python functional_tests.py
Traceback (most recent call last):
  File "functional_tests.py", line 10, in <module>
    assert 'To-Do' in browser.title
AssertionError
```

这就是**预期失败**。其实失败是好消息，虽不像测试通过那么令人振奋，但至少事出有因，证明测试编写得正确。

2.2 Python标准库中的unittest模块

上述测试中有几个恼人的问题需要处理。首先，“AssertionError”消息没什么用，如果测试能指出在浏览器的标题中到底找到了什么就好了。其次，Firefox 窗口一直停留在桌面上，如果能自动将其关闭就好了。

要解决第一个问题，可以使用 `assert` 关键字的第二个参数，写成：

```
assert 'To-Do' in browser.title, "Browser title was " + browser.title
```

Firefox 窗口可在 `try/finally` 语句中关闭。但这种问题在测试中很常见，标准库中的 `unittest` 模块已经提供了现成的解决方法。使用这种方法吧！在 functional_tests.py 中写入如下代码：

functional_tests.py

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):  ❶

    def setUp(self):  ❸
        self.browser = webdriver.Firefox()

    def tearDown(self):  ❸
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):  ❷
        # 伊迪丝听说有一个很酷的在线待办事项应用
        # 她去看了这个应用的首页
        self.browser.get('http://localhost:8000')

        # 她注意到网页的标题和头部都包含“To-Do”这个词
        self.assertIn('To-Do', self.browser.title)  ❹
        self.fail('Finish the test!')  ❺

        # 应用邀请她输入一个待办事项
        [其余的注释和之前一样]

if __name__ == '__main__':  ❻
    unittest.main(warnings='ignore')  ❼
```

你可能注意到以下几个地方了。

❶ 测试组织成类的形式，继承自 `unittest.TestCase`。

❷ 测试的主要代码写在名为 `test_can_start_a_list_and_retrieve_it_later` 的方法中。名字以 `test_` 开头的方法都是测试方法，由测试运行程序运行。类中可以定义多个测试方法。为测试方法起个有意义的名字是个好主意。

❸ `setUp` 和 `tearDown` 是特殊的方法，分别在各个测试方法之前和之后运行。我使用这两个方法打开和关闭浏览器。注意，这两个方法有点类似 `try/except` 语句，就算测试中出错了，也会运行 `tearDown` 方法。[1] 测试结束后，Firefox 窗口不会一直停留在桌面上了。

❹ 使用 `self.assertIn` 代替 `assert` 编写测试断言。`unittest` 提供了很多这种用于编写测试断言的辅助函数，如 `assertEqual`、`assertTrue` 和 `assertFalse` 等。更多断言辅助函数参见 `unittest` 的文档。

注 1：唯有一个特例：如果 `setUp` 方法抛出异常，`tearDown` 方法就不会运行。

❺ 不管怎样，`self.fail` 都会失败，生成指定的错误消息。我使用这个方法提醒测试结束了。

❻ 最后是 `if __name__ == '__main__'` 分句（如果你之前没见过这种用法，我告诉你，Python 脚本使用这个语句检查自己是否在命令行中运行，而不是在其他脚本中导入）。我们调用 `unittest.main()` 启动 `unittest` 的测试运行程序，这个程序会在文件中自动查找测试类和方法，然后运行。

❼ `warnings='ignore'` 的作用是禁止抛出 `ResourceWarning` 异常。写作本书时这个异常会抛出，但你阅读时我可能已经把这个参数去掉了。你可以把这个参数删掉，看一下效果。

> 如果你阅读 Django 关于测试的文档，可能会看到有个名为 `LiveServerTestCase` 的类，而且想知道我们现在能否使用它。你能阅读这份友好的手册真是值得表扬！目前来说，`LiveServerTestCase` 有点复杂，但我保证后面的章节会用到。

来试一下这个测试。

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "functional_tests.py", line 18, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'Welcome to Django'

 ---------------------------------------------------------------------
Ran 1 test in 1.747s

FAILED (failures=1)
```

这样是不是更好了？这个测试清理了 Firefox 窗口，显示了一个排版精美的报告，指出运行了几个测试，其中有几个测试失败了，而且 `assertIn` 还显示了一个有利于调试的错误消息。太棒了！

2.3　提交

现在是提交代码的好时机，因为已经做了一次完整的修改。我们扩展了功能测试，加入注释说明我们要在最简可用的待办事项清单应用中执行哪些操作。我们还使用 Python 中的 `unittest` 模块及其提供的各种测试辅助函数重写了测试。

执行 **`git status`** 命令，你会发现只有 functional_tests.py 文件的内容变化了。然后执行 **`git diff`** 命令，查看上一次提交和当前硬盘中保存内容之间的差异，你会发现 functional_tests.py 文件的变动很大：

```
$ git diff
diff --git a/functional_tests.py b/functional_tests.py
```

```
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
@@ -1,6 +1,45 @@
 from selenium import webdriver
+import unittest

-browser = webdriver.Firefox()
-browser.get('http://localhost:8000')
+class NewVisitorTest(unittest.TestCase):

-assert 'Django' in browser.title
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+
+    def tearDown(self):
+        self.browser.quit()
[...]
```

现在执行下述命令：

```
$ git commit -a
```

-a 的意思是：自动添加已跟踪文件（即已经提交的各文件）中的改动。上述命令不会添加全新的文件（你要使用 `git add` 命令手动添加这些文件）。不过就像这个例子一样，经常没有添加新文件，因此这是个很有用的简便用法。

弹出编辑器后，写入一个描述性的提交消息，比如“使用注释编写规格的首个功能测试，而且使用了 `unittest`”。

现在我们身处一个绝妙的位置，可以开始为这个清单应用编写真正的代码了。请继续往下阅读。

有用的 TDD 概念

- 用户故事

  从用户的角度描述应用应该如何运行。用来组织功能测试。

- 预期失败

  意料之中的失败。

第 3 章

使用单元测试测试简单的首页

上一章结束时功能测试是失败的，失败消息指出功能测试希望网站的首页标题中包含“To-Do”这个词。现在要开始编写这个应用了。

警告：要动真格的了

我故意把前两章写得这么友好和简单。从现在开始，要真正编写代码了。提前给你打个预防针：有些地方会出错。你看到的结果可能和我说的不一样。这是好事，因为这才是磨练意志的学习经历。

出现这种情况，一个可能的原因是，我表述不清，让你误解了我的本意。你要退一步想想要实现的是什么，在编辑哪个文件，想让用户做些什么，在测试什么，为什么要测试？有可能你编辑了错误的文件或函数，或者运行了其他测试。我觉得停下来想一想能更好地学习 TDD，比照搬所有操作、复制粘贴代码强得多。

还有一种原因，可能真有问题。认真阅读错误消息（详情请参见 3.5 节的“阅读调用跟踪”），你会找到原因的。可能是漏掉了一个逗号或末尾的斜线，或者某个 Selenium 查找方法少写了一个“s”。但是，正如 Zed Shaw 所说，这种调试也是学习过程的重要组成部分，所以一定要坚持到底！

如果你真的卡住了，随时可以给我发电子邮件，或者在谷歌小组（https://groups.google.com/forum/#!forum/obey-the-testing-goat-book）中发帖。祝你调试愉快！

3.1 第一个Django应用，第一个单元测试

Django 鼓励以**应用**的形式组织代码。这么做，一个项目中可以放多个应用，而且可以使用

其他人开发的第三方应用，也可以重用自己在其他项目中开发的应用。我承认，我自己从没真正这么做过。不过，应用的确是组织代码的好方法。

为待办事项清单创建一个应用：

```
$ python manage.py startapp lists
```

这个命令会在 superlists 文件夹中创建子文件夹 lists，与 superlists 子文件夹相邻，并在 lists 中创建一些占位文件，用来保存模型、视图以及目前最关注的测试：

```
superlists/
├── db.sqlite3
├── functional_tests.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

3.2 单元测试及其与功能测试的区别

正如给事物所贴的众多标签一样，单元测试和功能测试之间的界线有时不那么清晰。不过，二者之间有个基本区别：功能测试站在用户的角度从外部测试应用，单元测试则站在程序员的角度从内部测试应用。

我遵从的 TDD 方法同时使用这两种类型测试应用。采用的工作流程大致如下。

(1) 先写**功能测试**，从用户的角度描述应用的新功能。

(2) 功能测试失败后，想办法编写代码让它通过（或者说至少让当前失败的测试通过）。此时，使用一个或多个**单元测试**定义希望代码实现的效果，保证为应用中的每一行代码（至少）编写一个单元测试。

(3) 单元测试失败后，编写最少量的**应用代码**，刚好让单元测试通过。有时，要在第 2 步和第 3 步之间多次往复，直到我们觉得功能测试有一点进展为止。

(4) 然后，再次运行功能测试，看能否通过，或者有没有进展。这一步可能促使我们编写一些新的单元测试和代码等。

由此可以看出，这整个过程中，功能测试站在高层驱动开发，而单元测试则从低层驱动我们做些什么。

这个过程看起来是不是有点儿烦琐？有时确实如此，但功能测试和单元测试的目的不完全一样，而且最终写出的测试代码往往区别也很大。

> 功能测试的作用是帮助你开发具有所需功能的应用，还能保证你不会无意中破坏这些功能。单元测试的作用是帮助你编写简洁无错的代码。

理论讲得够多了，下面来看一下如何实践。

3.3　Django中的单元测试

来看一下如何为首页视图编写单元测试。打开新生成的文件 lists/tests.py，你会看到类似下面的内容：

lists/tests.py

```
from django.test import TestCase

# 在这里编写测试
```

Django 建议我们使用 `TestCase` 的一个特殊版本。这个版本由 Django 提供，是标准版 `unittest.TestCase` 的增强版，添加了一些 Django 专用的功能，后面几章会介绍。

你已经知道 TDD 循环要从失败的测试开始，然后再编写代码让其通过。但在此之前，不管单元测试的内容是什么，我们都想知道自动化测试运行程序能否运行我们编写的单元测试。直接运行 functional_tests.py。但 Django 生成的这个文件有点儿神奇，所以要确认一下。来故意编写一个会失败的愚蠢测试：

lists/tests.py

```
from django.test import TestCase

class SmokeTest(TestCase):

    def test_bad_maths(self):
        self.assertEqual(1 + 1, 3)
```

现在，要启动神奇的 Django 测试运行程序。和之前一样，这也是一个 `manage.py` 命令：

```
$ python manage.py test
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_bad_maths (lists.tests.SmokeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 6, in test_bad_maths
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

```
----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

太好了，看起来能运行单元测试。现在是提交的好时机：

```
$ git status  # 会显示一个消息，说没跟踪lists/
$ git add lists
$ git diff --staged  # 会显示将要提交的内容差异
$ git commit -m "Add app for lists, with deliberately failing unit test"
```

你猜得没错，`-m` 标志的作用是让你在命令行中编写提交消息，这样就不需要使用编辑器了。如何使用 Git 命令行取决于你自己，我只是向你展示我经常见到的用法。不管使用哪种方法，提交时要遵守一个主要原则：**提交前一定要审查想提交的内容**。

3.4 Django中的MVC、URL和视图函数

总的来说，Django 遵守了经典的**模型－视图－控制器**（Model-View-Controller，MVC）模式，**但并没严格遵守**。Django 确实有模型，但视图更像是控制器，模板其实才是视图。不过，MVC 的思想还在。如果你有兴趣，可以看一下 Django 常见问题解答（https://docs.djangoproject.com/en/1.11/faq/general/）中的详细说明。

抛开这些，Django 和任何一种 Web 服务器一样，主要任务是决定用户访问网站中的某个 URL 时做些什么。Django 的工作流程有点儿类似下述过程。

(1) 针对某个 URL 的 HTTP **请求**进入。
(2) Django 使用一些规则决定由哪个**视图**函数处理这个请求（这一步叫作**解析** URL）。
(3) 选中的视图函数处理请求，然后返回 HTTP **响应**。

因此要测试两件事。

- 能否解析网站根路径（“/”）的 URL，将其对应到我们编写的某个视图函数上？
- 能否让视图函数返回一些 HTML，让功能测试通过？

先编写第一个测试。打开 lists/tests.py，把之前编写的愚蠢测试改成如下代码：

lists/tests.py

```
from django.urls import resolve
from django.test import TestCase
from lists.views import home_page   ❷

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')   ❶
        self.assertEqual(found.func, home_page)   ❶
```

这段代码是什么意思呢？

❶ `resolve` 是 Django 内部使用的函数，用于解析 URL，并将其映射到相应的视图函数上。检查解析网站根路径“/”时，是否能找到名为 `home_page` 的函数。

❷ 这个函数是什么？这是接下来要定义的视图函数，其作用是返回所需的 HTML。从 `import` 语句可以看出，要把这个函数保存在文件 lists/views.py 中。

那么，你觉得运行这个测试后会有什么结果？

```
$ python manage.py test
ImportError: cannot import name 'home_page'
```

这个结果很容易预料，不过错误消息有点无趣：我们试图导入还没定义的函数。但是这对 TDD 来说算是好消息，预料之中的异常也算是预期失败。现在，功能测试和单元测试都失败了，在测试山羊的庇护下，我们可以编写代码了。

3.5　终于可以编写一些应用代码了

很兴奋吧？但要提醒一下：使用 TDD 时要耐着性子，步步为营。尤其是学习和起步阶段，一次只能修改（或添加）一行代码。每一次修改的代码要尽量少，让失败的测试通过即可。

我是故意这么极端的。还记得这个测试为什么会失败吗？因为我们无法从 `lists.views` 中导入 `home_page`。很好，来修正这个问题——仅修正这一个而已。在 lists/views.py 中写入下面的代码：

lists/views.py

```
from django.shortcuts import render

# 在这里编写视图
home_page = None
```

你可能会想：“不是开玩笑吧？”

我知道你会这么想，因为我的同事第一次给我演示 TDD 时我也是这么想的。耐心一点，稍后会分析这么做是否太极端。现在，就算你有点儿恼怒，也请跟着我一起做，看添加的这段代码能否帮助我们编写正确的代码，向前迈进一小步。

再次运行测试：

```
$ python manage.py test
Creating test database for alias 'default'...
E
======================================================================
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')
```

```
  File ".../django/urls/base.py", line 27, in resolve
    return get_resolver(urlconf).resolve(path)
  File ".../django/urls/resolvers.py", line 392, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}

 ---------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (errors=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

阅读调用跟踪

花点儿时间讲解如何阅读调用跟踪，因为在 TDD 中经常要做这件事。很快你就能学会如何扫视调用跟踪，找出解决问题的线索：

```
======================================================================
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)  ❷
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')  ❸
  File ".../django/urls/base.py", line 27, in resolve
    return get_resolver(urlconf).resolve(path)
  File ".../django/urls/resolvers.py", line 392, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver  ❶
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}  ❶
 ---------------------------------------------------------------------
[...]
```

❶ 首先应该查看**错误本身**。有时你只需查看这一处，就能立即找出问题所在。但某些时候，比如这个例子，原因就不是那么明显。

❷ 接下来要确认**哪个测试失败了**。是刚才编写按预期会失败的那个测试吗？在这个例子中，就是这个测试。

❸ 然后查看导致失败的**测试代码**。要从调用跟踪的顶部往下看，找出错误发生在哪个测试文件中哪个测试函数的哪一行代码。在这个例子中，错误发生在调用 `resolve` 函数解析“/”的那一行代码。

通常还有第四步，即继续往下看，查找问题牵涉的**应用代码**。在这个例子中全是 Django 的代码，不过在本书后面的内容中有很多示例都用到了这一步。

把以上几步的结果综合起来，可以解读出这个调用跟踪：尝试解析“/”时，Django 抛出了 404 错误。也就是说，Django 无法找到“/”的 URL 映射。下面来解决这个问题。

3.6 urls.py

测试表明，需要一个 URL 映射。Django 用 urls.py 文件把 URL 映射到视图函数上。在文件夹 superlists/superlists 中有个主 urls.py 文件，这个文件应用于整个网站。看一下其中的内容：

superlists/urls.py

```
"""superlists URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.11/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns:  url(r'^blog/', include('blog.urls'))
"""
from django.conf.urls import url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

和之前一样，这个文件中也有很多 Django 生成的辅助注释和默认建议。

`url` 条目的前半部分是正则表达式，定义适用于哪些 URL。后半部分说明把请求发往何处：发给导入的视图函数，或是别处的 urls.py 文件。

第一个示例条目使用的正则表达式是 `^$`，表示空字符串。这和网站的根路径，即我们要测试的“/”一样吗？分析一下，如果把这行代码的注释去掉会发生什么事呢？

> 如果你从未见过正则表达式，现在相信我所说的就行。不过你要记在心上，稍后去学习如何使用正则表达式。

我们还将去掉 admin URL，因为暂时用不到 Django 的管理后台。

superlists/urls.py

```
from django.conf.urls import url
from lists import views

urlpatterns = [
    url(r'^$', views.home_page, name='home'),
]
```

执行命令 **python manage.py test**，再次运行测试：

```
[...]
TypeError: view must be a callable or a list/tuple in the case of include().
```

确实有进展，不再显示 404 错误了。

调用跟踪有点乱，不过最后一行却指出了问题所在：单元测试把地址“/”和文件 lists/views.py 中的 `home_page = None` 连接起来了，现在测试抱怨 `home_page` 无法调用。由此我们知道，要调整一下，把 `home_page` 从 `None` 变成真正的函数。记住，每次改动代码都由测试驱使。

回到文件 lists/views.py，把内容改成：

lists/views.py

```
from django.shortcuts import render

# 在这里编写视图
def home_page():
    pass
```

现在测试的结果如何？

```
$ python manage.py test
Creating test database for alias 'default'...
.
 ----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

太好了，第一个测试终于通过了！这是一个重要时刻，我觉得值得提交一次：

```
$ git diff # 会显示urls.py、tests.py和views.py中的变动
$ git commit -am "First unit test and url mapping, dummy view"
```

这是我要介绍的最后一种 `git commit` 用法，把 `a` 和 `m` 标志放在一起使用，意思是添加所有已跟踪文件中的改动，而且使用命令行中输入的提交消息。

> `git commit -am` 是最快捷的方式，但关于提交内容的反馈信息最少，所以在此之前要先执行 `git status` 和 `git diff`，弄清楚要把哪些改动放入仓库。

3.7　为视图编写单元测试

该为视图编写测试了。此时，不能使用什么都不做的函数了，我们要定义一个函数，向浏览器返回真正的 HTML 响应。打开 lists/tests.py，添加一个新**测试方法**。我会解释每一行代码的作用。

lists/tests.py

```
from django.urls import resolve
from django.test import TestCase
from django.http import HttpRequest

from lists.views import home_page


class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)

    def test_home_page_returns_correct_html(self):
        request = HttpRequest()  ❶
        response = home_page(request)  ❷
        html = response.content.decode('utf8')  ❸
        self.assertTrue(html.startswith('<html>'))  ❹
        self.assertIn('<title>To-Do lists</title>', html)  ❺
        self.assertTrue(html.endswith('</html>'))  ❹
```

这个新测试方法都做了些什么呢？

❶ 创建了一个 `HttpRequest` 对象，用户在浏览器中请求网页时，Django 看到的就是 `HttpRequest` 对象。

❷ 把这个 `HttpRequest` 对象传给 `home_page` 视图，得到响应。听说响应对象是 `HttpResponse` 类的实例时，你应该不会觉得奇怪。

❸ 然后，提取响应的 `.content`。得到的结果是原始字节，即发给用户浏览器的 0 和 1。随后，调用 `.decode()`，把原始字节转换成发给用户的 HTML 字符串。

❹ 希望响应以 `<html>` 标签开头，并在结尾处关闭该标签。

❺ 希望响应中有一个 `<title>` 标签，其内容包含单词“To-Do lists”——因为在功能测试中做了这项测试。

再次说明，单元测试由功能测试驱动，而且更接近于真正的代码。编写单元测试时，按照程序员的方式思考。

运行单元测试，看看进展如何：

```
TypeError: home_page() takes 0 positional arguments but 1 was given
```

“单元测试/编写代码”循环

现在可以开始适应 TDD 中的**单元测试 / 编写代码**循环了。

(1) 在终端里运行单元测试，看它们是如何失败的。
(2) 在编辑器中改动最少量的代码，让当前失败的测试通过。

然后不断重复。

想保证编写的代码无误，每次改动的幅度就要尽量小。这么做才能确保每一部分代码都有对应的测试监护。

乍一看工作量很大，初期也的确如此。但熟练之后你便会发现，即使步伐迈得很小，编程的速度也很快。我们在工作中就是这样编写实际代码的。

看一下这个循环可以运转多快。

- 小幅代码改动：

lists/views.py

```
def home_page(request):
    pass
```

- 运行测试：

```
html = response.content.decode('utf8')
AttributeError: 'NoneType' object has no attribute 'content'
```

- 编写代码——如你所料，使用 `django.http.HttpResponse`：

lists/views.py

```
from django.http import HttpResponse

# 在这里编写视图
def home_page(request):
    return HttpResponse()
```

- 再运行测试：

```
    self.assertTrue(html.startswith('<html>'))
AssertionError: False is not true
```

- 再编写代码：

lists/views.py

```
def home_page(request):
    return HttpResponse('<html>')
```

- 运行测试：

```
AssertionError: '<title>To-Do lists</title>' not found in '<html>'
```

- 编写代码：

lists/views.py

```
def home_page(request):
    return HttpResponse('<html><title>To-Do lists</title>')
```

- 运行测试——快通过了吧？

```
    self.assertTrue(html.endswith('</html>'))
AssertionError: False is not true
```

- 加油，最后一击：

lists/views.py

```
def home_page(request):
    return HttpResponse('<html><title>To-Do lists</title></html>')
```

- 通过了吗？

```
$ python manage.py test
Creating test database for alias 'default'...
..
 ---------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

确实通过了。现在要运行功能测试。如果已经关闭了开发服务器，别忘了启动。感觉这是最后一次运行测试了吧，真是这样吗？

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "functional_tests.py", line 19, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!

 ---------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

失败了，怎么会？哦，原来是那个提醒？是吗？是的！我们成功编写了一个网页！

好吧，我觉得这样结束本章很刺激。你可能还有点儿摸不着头脑，或许还想知道怎么调整这些测试，别担心，后面的章节会讲。我只是想在临近收尾的时候让你兴奋一下。

要做一次提交，平复一下心情，再回想学到了什么：

```
$ git diff  # 会显示tests.py中的新测试方法，以及views.py中的视图
$ git commit -am "Basic view now returns minimal HTML"
```

这一章内容真丰富啊！为什么不执行 `git log` 命令回顾一下我们取得的进展呢？或许还可以指定 `--oneline` 标志：

```
$ git log --oneline
a6e6cc9 Basic view now returns minimal HTML
450c0f3 First unit test and url mapping, dummy view
ea2b037 Add app for lists, with deliberately failing unit test
[...]
```

不错，本章介绍了以下知识。

- 新建 Django 应用。
- Django 的单元测试运行程序。
- 功能测试和单元测试之间的区别。
- Django 解析 URL 的方法，urls.py 文件的作用。
- Django 的视图函数，请求和响应对象。
- 如何返回简单的 HTML。

有用的命令和概念

- 启动 Django 的开发服务器

  python manage.py runserver

- 运行功能测试

  python functional_tests.py

- 运行单元测试

  python manage.py test

- “单元测试 / 编写代码”循环

  (1) 在终端里运行单元测试。

  (2) 在编辑器中改动最少量的代码。

  (3) 重复上两步。

第 4 章

测试（及重构）的目的

现在已经实际演练了基本的 TDD 流程，该停下来说说为什么这么做了。

我想象得出很多读者心中都积压了一些挫败感，某些读者可能以前写过单元测试，另一些读者可能只想快速学会如何测试。你们心中有些疑问，比如说：

- 编写的测试是不是有点儿多了？
- 其中一些测试肯定有重复吧，比如功能测试和单元测试之间？
- 我的意思是，你为什么要在单元测试中导入 `django.core.urlresolvers` 呢？这不是在测试作为第三方代码的 Django 吗？我觉得没必要这么做，这么想对吗？
- 单元测试有点儿太琐碎了，测试一行声明代码，而且只让函数返回一个常量，这么做难道不是在浪费时间？我们是不是应该把时间腾出来为复杂功能编写测试？
- “单元测试 / 编写代码”循环中的小幅改动有必要吗？我们应该可以直接跳到最后一步吧？我想说，`home_page = None`？真的有必要吗？
- 难道现实中你**真这样**编写代码吗？

年轻人啊！以前我也这样满腹疑问。这些确实是很好的问题，其实，到现在我也经常问自己这些问题。真的值得做这些事吗？这么做是不是有点儿盲目？

4.1 编程就像从井里打水

编程其实很难，我们的成功往往得益于自己的聪明才智。假如我们不那么聪明，TDD 就能助我们一臂之力。Kent Beck（TDD 理念基本上就是他发明的）打了个比方。试想你用绳子从井里提一桶水，如果井不太深，而且桶不是很满，提起来很容易。就算提满满一桶水，刚开始也很容易。但要不了多久你就累了。TDD 理念好比是一个棘轮，你可以使用它保存当前的进度，休息一会儿，而且能保证进度绝不倒退。这样你就没必要**一直**那么聪明了（如图 4-1）。

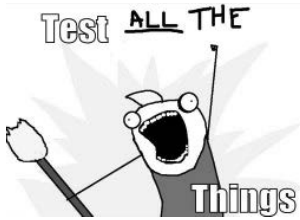


图 4-1：全部都要测试[1]

好吧，或许你**基本上**接受 TDD 是个好主意，但仍然认为我做得太极端了，有必要测试得这么细、步子这么小吗？

测试是一种**技能**，不是天生就会的。因为很多结果不会立刻显现，需要等待很长一段时间，所以目前你要强迫自己这么做。这就是测试山羊的图片想要表达的——你要对测试顽固一点儿。

细化测试每个函数的好处

就目前而言，测试简单的函数和常量看起来有点傻。

你可能觉得不遵守这么严格的规则，**漏掉一些**单元测试，应该也算得上是 TDD。但是在这本书中，我所演示的是完整而严格的 TDD 流程。像学习武术中的招式一样，在不受影响的可控环境中才能让技能变成肌肉记忆。现在看起来之所以琐碎，是因为我们刚开始举的例子很简单。程序变复杂后问题就来了，到时你就知道测试的重要性了。你要面临的危险是，复杂性逐渐靠近，而你可能没发觉，但不久之后你就会变成温水煮青蛙。

我赞成为简单的函数编写细化的简单测试，关于这一观点我还有这么两点要说。

首先，既然测试那么简单，写起来就不会花很长时间。所以，别抱怨了，只管写就是了。

其次，占位测试很重要。先为简单的函数写好测试，当函数变复杂后，这道心理障碍就容易迈过去。你可能会在函数中添加一个 `if` 语句，几周后再添加一个 `for` 循环，不知不觉间就将其变成一个基于元类（meta-class）的多态树状结构解析器了。因为从一开始你就编写了测试，每次修改都会自然而然地添加新测试，最终得到的是一个测试良好的函数。相反，如果你试图判断函数什么时候才复杂到需要编写测试的话，那就太主观了，而且情况会变得更糟，因为没有占位测试，此时开始编写测试需要投入很多精力，每次改动代码都冒着风险，你开始拖延，很快青蛙就煮熟了。

不要试图找一些不靠谱的主观规则，去判断什么时候应该编写测试，什么时候可以全身而退。我建议你现在遵守我制定的训练方法，因为所有技能都一样，只有花时间学会了规则才能打破规则。

接下来继续实践。

注 1：原画出自 Allie Brosh 的网站 Hyperbole and a Half。——译者注

4.2　使用Selenium测试用户交互

前一章结束时进展到哪里了？重新运行测试找出答案：

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "functional_tests.py", line 19, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!

 ---------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

你运行了吗？是不是看到一个错误，说**加载页面出错**或者**无法连接**？我也看到了。这是因为运行测试之前没有使用 `manage.py runserver` 启动开发服务器。运行这个命令，然后你会看到我们期待的那个失败消息。

> TDD 的优点之一是，永远不会忘记接下该做什么——重新运行测试就知道要做的事了。

失败消息说“Finish the test”（结束这个测试），那么就来结束它吧！打开 functional_tests.py 文件，扩充其中的功能测试：

functional_tests.py

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time
import unittest

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()

    def tearDown(self):
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):
        # 伊迪丝听说有一个很酷的在线待办事项应用
        # 她去看了这个应用的首页
        self.browser.get('http://localhost:8000')

        # 她注意到网页的标题和头部都包含“To-Do”这个词
```

```
self.assertIn('To-Do', self.browser.title)
header_text = self.browser.find_element_by_tag_name('h1').text  ❶
self.assertIn('To-Do', header_text)

# 应用邀请她输入一个待办事项
inputbox = self.browser.find_element_by_id('id_new_item')  ❶
self.assertEqual(
    inputbox.get_attribute('placeholder'),
    'Enter a to-do item'
)

# 她在一个文本框中输入了“Buy peacock feathers”（购买孔雀羽毛）
# 伊迪丝的爱好是使用假蝇做鱼饵钓鱼
inputbox.send_keys('Buy peacock feathers')  ❷

# 她按回车键后，页面更新了
# 待办事项表格中显示了“1: Buy peacock feathers”
inputbox.send_keys(Keys.ENTER)  ❸
time.sleep(1)  ❹

table = self.browser.find_element_by_id('id_list_table')
rows = table.find_elements_by_tag_name('tr')  ❶
self.assertTrue(
    any(row.text == '1: Buy peacock feathers' for row in rows)
)

# 页面中又显示了一个文本框，可以输入其他的待办事项
# 她输入了“Use peacock feathers to make a fly”（使用孔雀羽毛做假蝇）
# 伊迪丝做事很有条理
self.fail('Finish the test!')

# 页面再次更新，她的清单中显示了这两个待办事项
[...]
```

❶ 我们使用了 Selenium 提供的几个用来查找网页内容的方法：`find_element_by_tag_name`、`find_element_by_id` 和 `find_element`**`s`**`_by_tag_name`（注意有个 s，也就是说这个方法会返回多个元素）。

❷ 我们还使用了 `send_keys`，这是 Selenium 在输入框中输入内容的方法。

❸ `Keys` 类（别忘了导入）的作用是发送回车键等特殊的按键。[2]

❹ 按下回车键后页面会刷新。`time.sleep` 的作用是等待页面加载完毕，这样才能针对新页面下断言。这叫“显式等待”（特别简单，第 6 章将加以改进）。

> 小心 Selenium 中 `find_element_by...` 和 `find_elements_by...` 这两类函数的区别。前者返回一个元素，如果找不到就抛出异常；后者返回一个列表，这个列表可能为空。

注 2：这里可以直接使用字符串 `"\n"`，但因为 `Keys` 还能发送 Ctrl 等特殊的按键，所以我觉得有必要用一下 `Keys` 类。

还有，留意一下 any 函数，它是 Python 中的原生函数，却鲜为人知。不用我解释这个函数的作用了吧？使用 Python 编程就是这么惬意。

不过，如果你不懂 Python 的话，我告诉你，any 函数的参数是个**生成器表达式**（generator expression），类似于**列表推导**（list comprehension），但比它更为出色。你需要仔细研究这个概念。你可以搜索 Guido 名为“From List Comprehensions to Generator Expressions”的文章。读完之后你就会知道，这个函数可不仅仅是为了让编程惬意。

看一下测试进展如何：

```
$ python functional_tests.py
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: h1
```

解释一下，测试报错在页面中找不到 <h1> 元素。看一下如何在首页的 HTML 中加入这个元素。

大幅修改功能测试后往往有必要提交一次。初稿中我没这么做，想通之后就后悔了，可是已经和其他代码混在一起提交了。其实提交得越频繁越好：

```
$ git diff # 会显示对functional_tests.py的改动
$ git commit -am "Functional test now checks we can input a to-do item"
```

4.3　遵守“不测试常量”规则，使用模板解决这个问题

看一下 lists/tests.py 中的单元测试。现在，要查找特定的 HTML 字符串，但这不是测试 HTML 的高效方法。一般来说，单元测试的规则之一是**不测试常量**。以文本形式测试 HTML 很大程度上就是测试常量。

换句话说，如果有如下的代码：

```
wibble = 3
```

在测试中就不太有必要这么写：

```
from myprogram import wibble
assert wibble == 3
```

单元测试要测试的其实是逻辑、流程控制和配置。编写断言检测 HTML 字符串中是否有指定的字符序列，不是单元测试应该做的。

而且，在 Python 代码中插入原始字符串真的不是处理 HTML 的正确方式。我们有更好的方法，那就是使用模板。如果把 HTML 放在一个扩展名为 .html 的文件中，先不说其他好处，单就得到更好的句法高亮支持这一点而言也值了。Python 领域有很多模板框架，Django 有自己的模板系统，而且很好用。来使用这个模板系统吧。

4.3.1 使用模板重构

现在要做的是让视图函数返回完全一样的 HTML，但使用不同的处理方式。这个过程叫作重构，即**在功能不变的前提下**改进代码。

功能不变是最重要的。如果重构时添加了新功能，很可能会产生问题。重构本身也是一门学问，有专门的参考书——Martin Fowler 写的《重构》。

重构的首要原则是不能没有测试。幸好我们在做测试驱动开发，测试已经有了。检查一下测试能否通过，测试能通过才能保证重构前后的表现一致：

```
$ python manage.py test
[...]
OK
```

很好！先把 HTML 字符串提取出来写入单独的文件。新建用于保存模板的文件夹 lists/templates，然后新建文件 lists/templates/home.html，再把 HTML 写入这个文件[3]。

lists/templates/home.html

```
<html>
    <title>To-Do lists</title>
</html>
```

高亮显示的句法，漂亮多了！接下来修改视图函数：

lists/views.py

```
from django.shortcuts import render

def home_page(request):
    return render(request, 'home.html')
```

现在不自己构建 `HttpResponse` 对象了，转而使用 Django 中的 `render` 函数。这个函数的第一个参数是请求对象（原因稍后说明），第二个参数是渲染的模板名。Django 会自动在所有的应用目录中搜索名为 templates 的文件夹，然后根据模板中的内容构建一个 `HttpResponse` 对象。

> 模板是 Django 中一个很强大的功能，使用模板的主要优势之一是能把 Python 变量代入 HTML 文本。现在还没用到这个功能，不过后面的章节会用到。这就是为什么使用 `render` 和 `render_to_string`（稍后用到），而不用原生的 `open` 函数手动从硬盘中读取模板文件。

看一下模板是否起作用了：

```
$ python manage.py test
[...]
```

注 3：有些人喜欢使用和应用同名的子文件夹（即 lists/templates/lists），然后使用 lists/home.html 引用这个模板，这叫作“模板命名空间”。我觉得对小型项目来说使用模板命名空间太复杂了，不过在大型项目中可能有用武之地。详情参阅 Django 教程（https://docs.djangoproject.com/en/1.11/intro/tutorial03/#write-views-that-actually-do-something）。

```
======================================================================
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest)  ❷
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 17, in
test_home_page_returns_correct_html
    response = home_page(request)  ❸
  File "/.../superlists/lists/views.py", line 5, in home_page
    return render(request, 'home.html')  ❹
  File "/usr/local/lib/python3.6/dist-packages/django/shortcuts.py", line 48,
in render
    return HttpResponse(loader.render_to_string(*args, **kwargs),
  File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line
170, in render_to_string
    t = get_template(template_name, dirs)
  File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line
144, in get_template
    template, origin = find_template(template_name, dirs)
  File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line
136, in find_template
    raise TemplateDoesNotExist(name)
django.template.base.TemplateDoesNotExist: home.html  ❶

----------------------------------------------------------------------
Ran 2 tests in 0.004s
```

又遇到一次分析调用跟踪的机会。

❶ 先看错误是什么：测试无法找到模板。

❷ 然后确认是哪个测试失败：很显然是测试视图 HTML 的测试。

❸ 然后找到导致失败的是测试中的哪一行：调用 `home_page` 函数那行。

❹ 最后，在应用的代码中找到导致失败的部分：调用 `render` 函数那段。

那为什么 Django 找不到模板呢？模板在 lists/templates 文件夹中，它就该放在这个位置啊。

原因是还没有**正式**在 Django 中注册 lists 应用。执行 `startapp` 命令以及在项目文件夹中存放一个应用还不够，你要告诉 Django **确实**要开发一个应用，并把这个应用添加到文件 settings.py 中。这么做才能保证万无一失。打开 settings.py，找到变量 `INSTALLED_APPS`，把 `lists` 加进去：

superlists/settings.py

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
]
```

可以看出，默认已经有很多应用了。只需把 `lists` 加到列表的末尾。别忘了在行尾加上逗号，这么做虽然不是必须的，但如果忘了，Python 会把不在同一行的两个字符串连起来，到时你就傻眼了。

现在可以再运行测试看看：

```
$ python manage.py test
    [...]
    self.assertTrue(html.endswith('</html>'))
AssertionError: False is not true
```

糟糕，还是无法通过。

> 你能否看到这个错误，取决于你使用的文本编辑器是否会在文件的最后添加一个空行。如果没看到，你可以跳过下面几段，直接跳到测试通过那部分。

不过确实有进展。看起来测试找到模板了，但最后三个断言失败了。很显然输出的末尾出了问题。我使用 `print (repr(html))` 调试这个问题，发现是因为转用模板后在响应的末尾引入了一个额外的空行（`\n`）。按下面的方式修改可以让测试通过：

lists/tests.py

```
self.assertTrue(html.strip().endswith('</html>'))
```

这么做有点像作弊，不过 HTML 文件末尾的空白并不重要。再运行测试看看：

```
$ python manage.py test
[...]
OK
```

对代码的重构结束了，测试也证实了重构前后的表现一致。现在可以修改测试，不再测试常量，检查是否渲染了正确的模板。

4.3.2　Django测试客户端

测试是否正确渲染模板的一种方法是在测试中手动渲染模板，然后与视图返回的结果做比较。为此，可以利用 Django 提供的 `render_to_string` 函数：

lists/tests.py

```
from django.template.loader import render_to_string
[...]

    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        response = home_page(request)
        html = response.content.decode('utf8')
        expected_html = render_to_string('home.html')
        self.assertEqual(html, expected_html)
```

但这样测试有点笨拙，而且转来转去调用 `.decode()` 和 `.strip()` 太牵扯精力。其实，Django 提供的测试客户端（Test Client）才是检查使用哪个模板的原生方式。相应的测试如下所示：

lists/tests.py

```
    def test_home_page_returns_correct_html(self):
        response = self.client.get('/')  ❶

        html = response.content.decode('utf8')  ❷
        self.assertTrue(html.startswith('<html>'))
        self.assertIn('<title>To-Do lists</title>', html)
        self.assertTrue(html.strip().endswith('</html>'))

        self.assertTemplateUsed(response, 'home.html')  ❸
```

❶ 不再手动创建 `HttpRequest` 对象，也不再直接调用视图函数，而是调用 `self.client.get`，并传入要测试的 URL。

❷ 暂时保留这一行，确保一切与之前一样正常。

❸ `.assertTemplateUsed` 是 Django `TestCase` 类提供的测试方法，用于检查响应是使用哪个模板渲染的（注意，这个方法只能测试通过测试客户端获取的响应）。

这个测试依然能通过：

```
Ran 2 tests in 0.016s

OK
```

我对没失败的测试始终有所怀疑，所以故意做点破坏：

lists/tests.py

```
        self.assertTemplateUsed(response, 'wrong.html')
```

这样还能看看错误消息是什么：

```
AssertionError: False is not true : Template 'wrong.html' was not a template
used to render the response. Actual template(s) used: home.html
```

消息的内容很有帮助！现在把断言改回去，顺便把旧的断言删掉。此外，还可以把原来的 `test_root_url_resolves` 测试删除，因为 Django 测试客户端已经隐式测试过了。我们把两个冗长的测试精简成了一个！

lists/tests.py (ch04l010)

```
from django.test import TestCase

class HomePageTest(TestCase):

    def test_uses_home_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')
```

注意，这里的重点是“不要测试常量，而应该测试实现方式”。很好！[4]

注 4：你是不是发现某些代码清单旁有 ch04l0xx 这样的文本，而且还不知道那是什么意思？这是本书示例仓库中特定提交（https://github.com/hjwp/book-example/commits/chapter_philosophy_and_refactoring）的引用，为本书中的测试（https://github.com/hjwp/Book-TDD-Web-Dev-Python/tree/master/tests）使用，也就是这本关于测试的书中的测试的测试；显然，测试自身也需要测试。

为什么不一直使用 Django 测试客户端？

你可能会问：“为什么不从一开始就使用 Django 测试客户端呢？”在现实中，我确实会这么做。但鉴于一些原因，我想告诉你如何“手动”实现。首先，这样可以逐个介绍相关概念，尽量使学习曲线保持平缓；其次，你可能不会一直使用 Django 构建应用，而且相关的测试工具也不是永远可用——但是直接调用函数，然后检查响应是永远可以采用的方法。

此外，Django 测试客户端也有不足之处，后文会讨论完全隔离的单元测试与由测试客户端推动向前的“整合”测试之间的区别。但就目前而言，测试客户端尚且算是务实的选择。

4.4 关于重构

这个重构的例子很烦琐。但正如 Kent Beck 在 *Test-Driven Development: By Example* 一书中所说的：“我是推荐你在实际工作中这么做吗？不是。我只是建议你要知道**怎么按照**这种方式做。”

其实，写这一部分时我的第一反应是先修改代码，直接使用 `assertTemplateUsed` 函数，删除那三个多余的断言，只在渲染得到的结果中检查期望看到的内容，然后再修改代码。但要注意，如果真这么做了可能就会犯错，因为我可能不会在模板中编写正确的 `<html>` 和 `<title>` 标签，而是随便写一些字符串。

重构时，修改代码或者测试，但不能同时修改。

在重构的过程中总有超前几步的冲动，想对应用的功能做些改动，不久，修改的文件就会变得越来越多，最终你会忘记自己身在何处，而且一切都无法正常运行了。如果你不想让自己变成“重构猫”（如图 4-2），迈的步子就要小一点，把重构和功能调整完全分开来做。

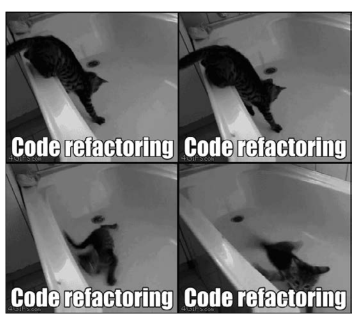


图 4-2：重构猫——记得要看完整的动态 GIF 图

> 后面会再次遇到重构猫，用来说明头脑发热、一次修改很多内容带来的后果。你可以把这只猫想象成卡通片中浮现在另一个肩膀上的魔鬼，它和测试山羊的观点是对立的，总给些不好的建议。

重构后最好做一次提交：

```
$ git status # 会看到tests.py、views.py、settings.py以及新建的templates文件夹
$ git add . # 还会添加尚未跟踪的templates文件夹
$ git diff --staged # 审查我们想提交的内容
$ git commit -m "Refactor home page view to use a template"
```

4.5 接着修改首页

现在功能测试还是失败的。修改代码，让它通过。因为 HTML 现在保存在模板中，可以尽情修改，无须编写额外的单元测试。我们需要一个 <h1> 元素：

lists/templates/home.html

```
<html>
    <head>
        <title>To-Do lists</title>
    </head>
    <body>
        <h1>Your To-Do list</h1>
    </body>
</html>
```

看一下功能测试是否认同这次修改：

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]
```

不错，继续修改：

lists/templates/home.html

```
    [...]
        <h1>Your To-Do list</h1>
        <input id="id_new_item" />
    </body>
    [...]
```

现在呢？

```
AssertionError: '' != 'Enter a to-do item'
```

加上占位文字：

lists/templates/home.html

```
    <input id="id_new_item" placeholder="Enter a to-do item" />
```

得到了下述错误：

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]
```

因此要在页面中加入表格。目前表格是空的：

lists/templates/home.html

```
    <input id="id_new_item" placeholder="Enter a to-do item" />
    <table id="id_list_table">
    </table>
</body>
```

现在功能测试的结果如何？

```
  File "functional_tests.py", line 43, in
test_can_start_a_list_and_retrieve_it_later
    any(row.text == '1: Buy peacock feathers' for row in rows)
AssertionError: False is not true
```

有点儿晦涩。可以使用行号找出问题所在，原来是前面我沾沾自喜的那个 `any` 函数导致的，或者更准确地说是 `assertTrue`，因为没有提供给它明确的失败消息。可以把自定义的错误消息传给 `unittest` 中的大多数 `assertX` 方法：

functional_tests.py

```
        self.assertTrue(
            any(row.text == '1: Buy peacock feathers' for row in rows),
            "New to-do item did not appear in table"
        )
```

再次运行功能测试，应该会看到我们编写的消息：

```
AssertionError: False is not true : New to-do item did not appear in table
```

不过现在如果想让测试通过，就要真正处理用户提交的表单，但这是下一章的话题。

现在做个提交吧：

```
$ git diff
$ git commit -am "Front page HTML now generated from a template"
```

多亏这次重构，视图能渲染模板了，也不再测试常量了，现在准备好处理用户的输入了。

4.6 总结：TDD流程

至此，我们已经在实践中见识了 TDD 流程中涉及的所有主要概念。

- 功能测试。
- 单元测试。
- “单元测试 / 编写代码”循环。
- 重构。

现在要稍微总结一下，或许可以画个流程图。请原谅我，做了这么多年管理顾问，养成了这个习惯。不过流程图也有好的一面，能清楚地表明流程中的循环。

TDD 的总体流程是什么呢？参见图 4-3。

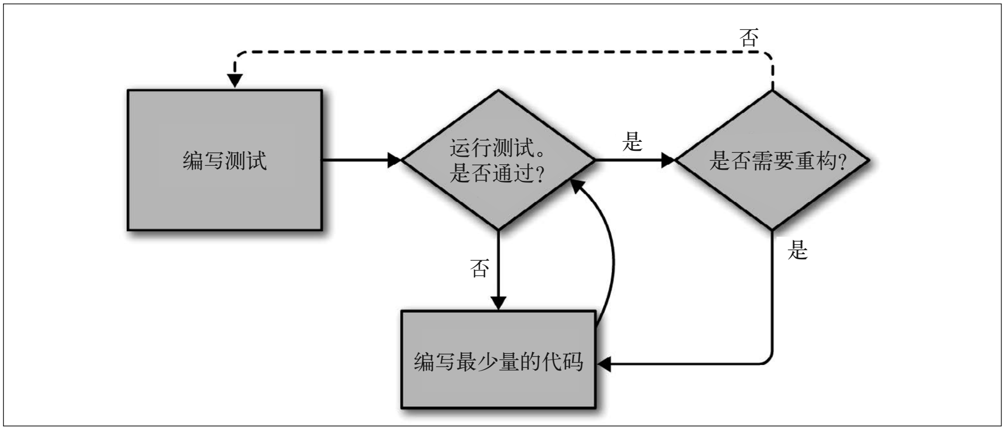


图 4-3：TDD 的总体流程

首先编写一个测试，运行这个测试看着它失败。然后编写最少量的代码取得一些进展，再运行测试。如此不断重复，直到测试通过为止。最后，或许还要重构代码，测试能确保不破坏任何功能。

如果既有功能测试，又有单元测试，该怎么运用这个流程呢？你可以把功能测试当作循环的一种高层视角，而“编写代码让功能测试通过”这一步则是另一个小型 TDD 循环，这个小循环使用单元测试，如图 4-4 所示。

编写一个功能测试，看着它失败。接下来，“编写代码让功能测试通过”这一步是一个小型 TDD 循环：编写一个或多个单元测试，然后进入“单元测试 / 编写代码”循环，直到单元测试通过为止。然后回到功能测试，查看是否有进展。这一步还可以多编写一些应用代码，再编写更多的单元测试，如此一直循环下去。

涉及功能测试时应该怎么重构呢？这就要使用功能测试检查重构前后的表现是否一致。不过，你可以修改、添加或删除单元测试，或者使用单元测试循环修改实现方式。

功能测试是应用能否正常运行的最终评判，而单元测试只是整个开发过程中的一个辅助工具。

这种看待事物的方式有时叫作“双循环测试驱动开发”。本书的优秀技术审校人员之一 Emily Bache 写了一篇博客，从不同的视角讨论了这个话题，推荐你阅读，名为“Coding Is Like Cooking”。

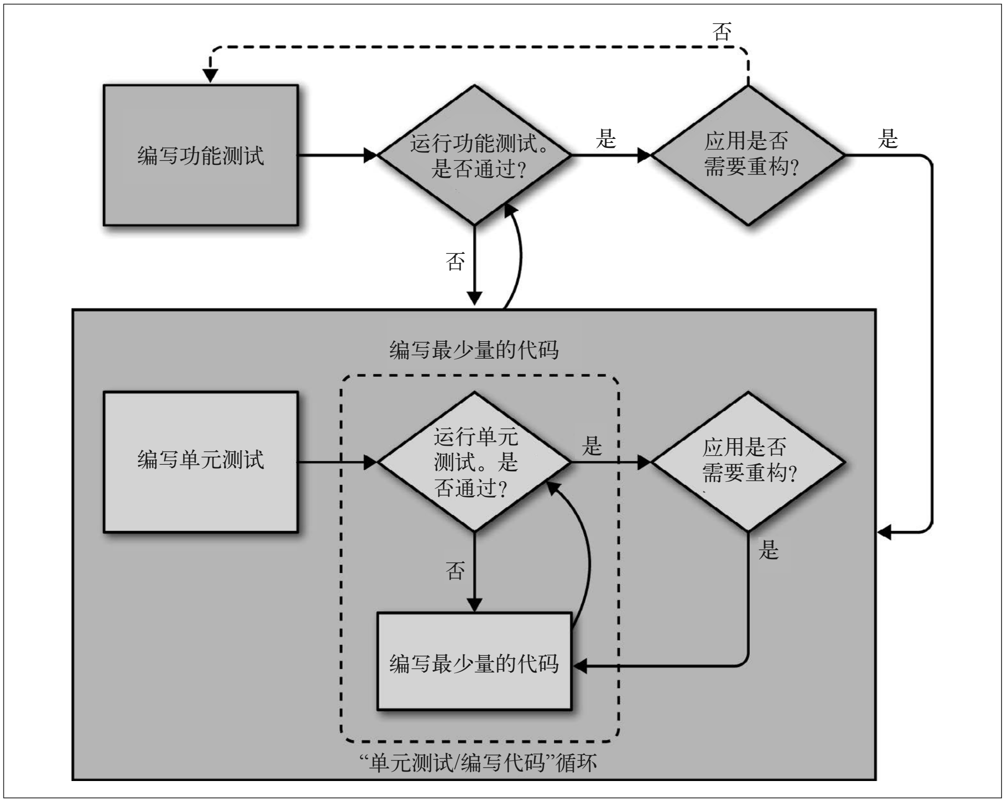


图 4-4：包含功能测试和单元测试的 TDD 流程

接下来的章节会更深入地探索这个工作流程中的各个组成部分。

如何检查你的代码，以及在必要时跳着阅读

书中使用的所有代码示例都可以到我放在 GitHub 中的仓库（https://github.com/hjwp/book-example/）中获取。因此，如果你想拿自己的代码和我的比较，可以到这个仓库中看一下。

每一章的代码都放在单独的分支中，各分支采用简短形式命名，比如说本章的示例在 chapter_philosophy_and_refactoring 这个分支中。这是本章结束时代码的快照。

所有分支的代码请至 http://www.ituring.com.cn/book/2052 下载。附录 J 说明了如何使用 Git 比较你我的代码。

第 5 章

保存用户输入：测试数据库

要获取用户输入的待办事项，发送给服务器，这样才能使用某种方式保存待办事项，然后再显示给用户查看。

刚开始写这一章时，我立即采用了我认为正确的设计方式：为清单和待办事项创建几个模型，为新建清单和待办事项创建一组不同的 URL，编写三个新视图函数，又为这些操作编写六七个新的单元测试。不过我还是忍住了没这么做。虽然我十分确定自己很聪明，能一次处理所有问题，但是 TDD 的重要思想是必要时一次只做一件事。所以我决定放慢脚步，每次只做必要的操作，让功能测试向前迈出一小步即可。

这么做是为了演示 TDD 对迭代式开发方法的支持——这种方法不是最快的，但最终仍能把你带到目的地。使用这种方法还有个不错的附带好处：**我可以一次只介绍一个新概念**，例如模型、处理 POST 请求和 Django 模板标签等，不必一股脑儿全抛给你。

并不是说你**不能**事先考虑后面的事，或者**不能**发挥自己的聪明才智。下一章我们会稍微多使用一点儿设计和预见思维，展示如何在 TDD 过程中运用这些思维方式。不过现在，我们要坚持自己是无知的，测试让做什么就做什么。

5.1　编写表单，发送POST请求

上一章末尾，测试指出无法保存用户的输入。现在，要使用标准的 HTML POST 请求。虽然有点无聊，但发送过程很简单。后文我们会见识到各种有趣的 HTML5 和 JavaScript 用法。

为了让浏览器发送 POST 请求，我们要做两件事。

(1) 给 `<input>` 元素指定 `name=` 属性。
(2) 把它放在 `<form>` 标签中，并为 `<form>` 标签指定 `method="POST"` 属性。

据此调整一下 lists/templates/home.html 中的模板：

lists/templates/home.html

```
<h1>Your To-Do list</h1>
<form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
</form>

<table id="id_list_table">
```

现在运行功能测试，会看到一个晦涩难懂、预料之外的错误：

```
$ python functional_tests.py
[...]
Traceback (most recent call last):
  File "functional_tests.py", line 40, in
test_can_start_a_list_and_retrieve_it_later
    table = self.browser.find_element_by_id('id_list_table')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]
```

如果功能测试出乎意料地失败了，可以做下面几件事，找出问题所在。

- 添加 `print` 语句，输出页面中当前显示的文本是什么。
- 改进**错误消息**，显示当前状态的更多信息。
- 亲自手动访问网站。
- 在测试执行过程中使用 `time.sleep` 暂停。

本书会分别介绍这几种调试方法，不过我发现自己经常使用 `time.sleep`。下面试一下这种方法。

其实，在错误发生之前就已经休眠了。那就延长休眠时间：

functional_tests.py

```
# 按回车键后，页面更新了
# 待办事项表格中显示了“1: Buy peacock feathers”
inputbox.send_keys(Keys.ENTER)
time.sleep(10)

table = self.browser.find_element_by_id('id_list_table')
```

如果 Selenium 运行得很慢，你可能已经发现了这一问题。现在再次运行功能测试，就有机会看看到底发生了什么：你会看到一个如图 5-1 所示的页面，显示了 Django 提供的很多调试信息。

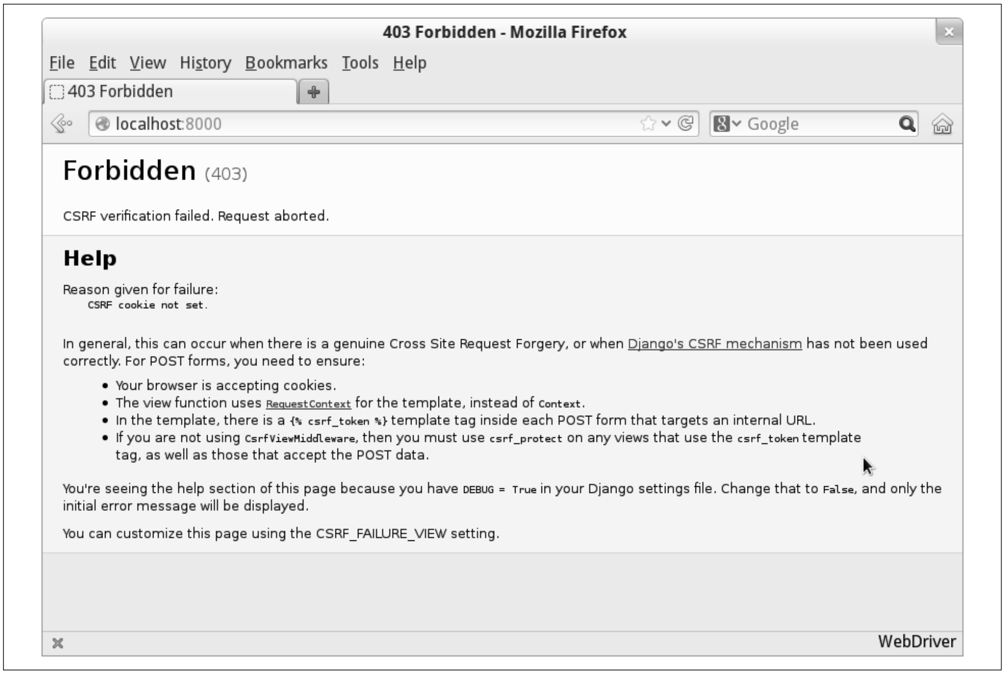


图 5-1：Django 中的调试页面，显示有 CSRF 错误

> **安全：异常有趣！**
>
> 如果你从未听说过**跨站请求伪造**（Cross-Site Request Forgery，CSRF）漏洞，现在就去查资料吧。和所有安全漏洞一样，研究起来很有趣。CSRF 是一种不寻常的、使用系统的巧妙方式。
>
> 在大学攻读计算机科学学位时，出于责任感，我报名学习了安全课程单元。**这个单元可能很枯燥乏味，但我觉得最好还是学一下**。结果证明，这是所有课程中最吸引人的单元，充满了黑客的乐趣，你要在特定的心境下思考如何通过意想不到的方式使用系统。
>
> 我要推荐学这门课程时使用的课本，Ross Anderson 写的 *Security Engineering*。这本书没有深入讲解纯粹的加密机制，而是讨论了很多意料之外的话题，例如开锁、伪造银行票据和喷墨打印机墨盒的经济原理，以及如何使用重放攻击（replay attack）戏弄南非空军的飞机等。这是本大部头书，大约 3 英寸厚，但相信我，绝对值得一读。

Django 针对 CSRF 的保护措施是在生成的每个表单中放置一个自动生成的令牌，通过这个令牌判断 POST 请求是否来自同一个网站。之前的模板都是纯粹的 HTML，在这里要首次体验 Django 模板的魔力，使用**模板标签**（template tag）添加 CSRF 令牌。模板标签的句法是花括号和百分号形式，即 `{% ... %}`——这种写法很有名，要连续多次同时按两个键，是世界上最麻烦的输入方式。

lists/templates/home.html

```
<form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
    {% csrf_token %}
</form>
```

渲染模板时，Django 会把这个模板标签替换成一个 `<input type="hidden">` 元素，其值是 CSRF 令牌。现在运行功能测试，会看到一个预期失败：

```
AssertionError: False is not true : New to-do item did not appear in table
```

因为 `time.sleep` 还在，所以测试会在最后一屏上暂停。可以看到，提交表单后新添加的待办事项不见了，页面刷新后又显示了一个空表单。这是因为还没连接服务器让它处理 POST 请求，所以服务器忽略请求，直接显示常规首页。

其实，现在可以删掉 `time.sleep` 了：

functional_tests.py

```
        # 待办事项表格中显示了“1: Buy peacock feathers”
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)

        table = self.browser.find_element_by_id('id_list_table')
```

5.2 在服务器中处理POST请求

还没为表单指定 `action=` 属性，因此提交表单后默认返回之前渲染的页面（即“/”），这个页面由视图函数 `home_page` 处理。下面修改这个视图函数，让它能处理 POST 请求。

这意味着要为视图函数 `home_page` 编写一个新的单元测试。打开文件 lists/tests.py，在 `HomePageTest` 类中添加一个新方法：

lists/tests.py (ch05l005)

```
    def test_uses_home_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')

    def test_can_save_a_POST_request(self):
        response = self.client.post('/', data={'item_text': 'A new list item'})
        self.assertIn('A new list item', response.content.decode())
```

为了发送 POST 请求，我们调用 `self.client.post`，传入 `data` 参数，指定想发送的表单数据。然后再检查 POST 请求渲染得到的 HTML 中是否有指定的文本。运行测试后，会看到预期的失败：

```
$ python manage.py test
[...]
AssertionError: 'A new list item' not found in '<html>\n    <head>\n
<title>To-Do lists</title>\n    </head>\n    <body>\n        <h1>Your To-Do
list</h1>\n        <form method="POST">\n            <input name="item_text"
```

```
[...]
</body>\n</html>\n'
```

为了让测试通过，可以添加一个 `if` 语句，为 POST 请求提供一个不同的代码执行路径。按照典型的 TDD 方式，先故意编写一个愚蠢的返回值：

lists/views.py

```
from django.http import HttpResponse
from django.shortcuts import render

def home_page(request):
    if request.method == 'POST':
        return HttpResponse(request.POST['item_text'])
    return render(request, 'home.html')
```

这样单元测试就能通过了，但这并不是我们真正想要做的。我们真正想要做的是把 POST 请求提交的数据添加到首页模板的表格里。

5.3　把Python变量传入模板中渲染

前面已经粗略展示了 Django 的模板句法，现在是时候领略它的真正强大之处了，即从视图的 Python 代码中把变量传入 HTML 模板。

先介绍在模板中使用哪种句法引入 Python 对象。要使用的符号是 `{{ ... }}`，它会以字符串的形式显示对象：

lists/templates/home.html

```
<body>
    <h1>Your To-Do list</h1>
    <form method="POST">
        <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
        {% csrf_token %}
    </form>

    <table id="id_list_table">
        <tr><td>{{ new_item_text }}</td></tr>
    </table>
</body>
```

这里要调整单元测试，检查是否依然使用这个模板：

lists/tests.py

```
    def test_can_save_a_POST_request(self):
        response = self.client.post('/', data={'item_text': 'A new list item'})
        self.assertIn('A new list item', response.content.decode())
        self.assertTemplateUsed(response, 'home.html')
```

跟预期一样，这个测试会失败：

```
AssertionError: No templates used to render the response
```

很好，故意编写的愚蠢返回值已经骗不过测试了，因此要重写视图函数，把 POST 请求中的参数传入模板。render 函数的第三个参数是一个字典，把模板变量的名称映射在值上：

lists/views.py (ch05l009)

```
def home_page(request):
    return render(request, 'home.html', {
        'new_item_text': request.POST['item_text'],
    })
```

然后再运行单元测试：

```
ERROR: test_uses_home_template (lists.tests.HomePageTest)
[...]
  File "/.../superlists/lists/views.py", line 5, in home_page
    'new_item_text': request.POST['item_text'],
[...]
django.utils.datastructures.MultiValueDictKeyError: "'item_text'"
```

看到的是**意料之外的失败**。

如果你记得阅读调用跟踪的方法，就会发现这次失败其实发生在**另一个**测试中。我们让正在处理的测试通过了，但是这个单元测试却导致了一个意想不到的结果，或者称之为“回归”：破坏了没有 POST 请求时执行的那条代码路径。

这就是测试的要义所在。不错，发生这样的事是可以预料的，但如果运气不好或者没有注意到呢？这时测试就能避免破坏应用功能，而且，因为我们在使用 TDD，所以能立即发现什么地方有问题。无须等待质量保证团队的反馈，也不用打开浏览器自己动手在网站中点来点去，直接就能修正问题。这次失败的修正方法如下：

lists/views.py

```
def home_page(request):
    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', ''),
    })
```

如果不理解这段代码，可以查阅 dict.get 的文档。

这个单元测试现在应该可以通过了。看一下功能测试的结果如何：

```
AssertionError: False is not true : New to-do item did not appear in table
```

> 如果你现在或者在本章其他地方看到的功能测试报错与这里不同，而是与 StaleElementReferenceException 有关，你可能就需要增加 time.sleep 的显式等待时间了——可以试试把 1 改成 2 或 3。更可靠的解决方法参见下一章。

错误消息没太大帮助。使用另一种功能测试的调试技术：改进错误消息。这或许是最有建设性的技术，因为改进后的错误消息一直存在，可以协助调试以后出现的错误：

functional_tests.py (ch05l011)

```
self.assertTrue(
    any(row.text == '1: Buy peacock feathers' for row in rows),
    f"New to-do item did not appear in table. Contents were:\n{table.text}"   ❶
)
```

❶ 你以前可能没有见过这种句法，这是 Python 新的 f 字符串句法（算是 Python 3.6 最令人激动的新特性）。只需在字符串前面加上一个 f，你就能使用花括号插入局部变量。详情请参见 Python 3.6 的版本说明。

改进后，测试给出了更有用的错误消息：

```
AssertionError: False is not true : New to-do item did not appear in table.
Contents were:
Buy peacock feathers
```

知道怎么改效果更好吗？稍微让断言别那么灵巧。你可能还记得，函数 `any` 让我很满意，但预览版的一个读者（感谢 Jason）建议我使用一种更简单的实现方式，把六行 `assertTrue` 换成一行 `assertIn`：

functional_tests.py (ch05l012)

```
    self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
```

这样好多了。自作聪明时一定要小心，因为你可能把问题**过度复杂化**了。修改之后自动获得了下面的错误消息：

```
    self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
AssertionError: '1: Buy peacock feathers' not found in ['Buy peacock feathers']
```

就当这是我应得的惩罚吧。

> 如果功能测试指出的错误是表格为空（not found in []），那就检查一下 `<input>` 标签，看看有没有正确设定 `name="item_text"` 属性。如果没有这个属性，用户的输入就不能与 `request.POST` 中正确的键关联。

上述错误消息的意思是，功能测试在枚举列表中的项目时希望第一个项目以“1:”开头。让测试通过最快的方法是修改模板时“作弊”：

lists/templates/home.html

```
    <tr><td>1: {{ new_item_text }}</td></tr>
```

“遇红 / 变绿 / 重构”和三角法

“单元测试 / 编写代码”循环有时也叫“**遇红 / 变绿 / 重构**”。

- 先写一个会失败的单元测试（**遇红**）。
- 编写尽可能简单的代码让测试通过（**变绿**），**就算作弊也行**。
- **重构**，改进代码，让其更合理。

那么，在重构阶段应该做些什么呢？如何判断什么时候应该把作弊的代码改成令我们满意的实现方式呢？

一种方法是**消除重复**：如果测试中使用了神奇常量（例如列表项目前面的“1:”），而且应用代码中也用了这个常量，这就算是重复，此时就应该重构。把神奇常量从应用代码中删掉往往意味着你不能再作弊了。

我觉得这种方法有点不太明确，所以经常使用第二种方法，这种方法叫作“**三角法**”：如果编写无法让你满意的作弊代码（例如返回一个神奇的常量）就能让测试通过，就**再写一个测试**，强制自己编写更好的代码。现在就要使用这种方法，扩充功能测试，检查输入的**第二个**列表项目中是否包含“2:”。

现在功能测试能执行到 `self.fail('Finish the test!')` 了。如果扩充功能测试，检查表格中添加的第二个待办事项（复制粘贴是好帮手），我们会发现刚才使用的简单处理方式不奏效了：

functional_tests.py

```
        # 页面中还有一个文本框，可以输入其他的待办事项
        # 她输入了“Use peacock feathers to make a fly”（使用孔雀羽毛做假蝇）
        # 伊迪丝做事很有条理
        inputbox = self.browser.find_element_by_id('id_new_item')
        inputbox.send_keys('Use peacock feathers to make a fly')
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)

        # 页面再次更新，清单中显示了这两个待办事项
        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')
        self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
        self.assertIn(
            '2: Use peacock feathers to make a fly',
             [row.text for row in rows]
        )

        # 伊迪丝想知道这个网站是否会记住她的清单
        # 她看到网站为她生成了一个唯一的URL
        # 页面中有一些文字解说这个功能
        self.fail('Finish the test!')

        # 她访问那个URL，发现待办事项清单还在
```

很显然，这个功能测试会返回一个错误：

```
AssertionError: '1: Buy peacock feathers' not found in ['1: Use peacock
feathers to make a fly']
```

5.4　事不过三，三则重构

在继续之前，先看一下功能测试中的**代码异味**。[1] 检查清单表格中新添加的待办事项时，用了三个几乎一样的代码块。编程中有个原则叫作**不要自我重复**（Don't Repeat Yourself，DRY），按照真言"**事不过三，三则重构**"的说法，运用这个原则。复制粘贴一次，可能还不用删除重复，但如果复制粘贴了三次，就该删除重复了。

要先提交目前已编写的代码。虽然网站还有重大瑕疵（只能处理一个待办事项），但仍然取得了一定进展。这些代码可能要全部重写，也可能不用，不管怎样，重构之前一定要提交：

```
$ git diff
# 会看到functional_tests.py、home.html、tests.py和views.py中的变动
$ git commit -a
```

然后重构功能测试。可以定义一个行间函数，不过这样会稍微搅乱测试流程，还是用辅助方法吧。记住，只有名字以 `test_` 开头的方法才会作为测试运行，可以根据需求使用其他方法。

functional_tests.py

```
    def tearDown(self):
        self.browser.quit()

    def check_for_row_in_list_table(self, row_text):
        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')
        self.assertIn(row_text, [row.text for row in rows])


    def test_can_start_a_list_and_retrieve_it_later(self):
        [...]
```

我喜欢把辅助方法放在类的顶部，置于 `tearDown` 和第一个测试之间。下面在功能测试中使用这个辅助方法：

functional_tests.py

```
    # 她按回车键后，页面更新了
    # 待办事项表格中显示了"1: Buy peacock feathers"
    inputbox.send_keys(Keys.ENTER)
    time.sleep(1)
    self.check_for_row_in_list_table('1: Buy peacock feathers')

    # 页面中又显示了一个文本框，可以输入其他的待办事项
    # 她输入了"Use peacock feathers to make a fly"（使用孔雀羽毛做假蝇）
    # 伊迪丝做事很有条理
    inputbox = self.browser.find_element_by_id('id_new_item')
    inputbox.send_keys('Use peacock feathers to make a fly')
    inputbox.send_keys(Keys.ENTER)
```

注 1：如果你没遇到过这个概念，我告诉你，"代码异味"表明一段代码需要重写。Jeff Atwood 在他的博客 Coding Horror 中搜集了很多这方面的资料。编程经验越丰富，你的鼻子就会变得越灵敏，能够嗅出代码中的异味。

```
    time.sleep(1)

    # 页面再次更新，她的清单中显示了这两个待办事项
    self.check_for_row_in_list_table('1: Buy peacock feathers')
    self.check_for_row_in_list_table('2: Use peacock feathers to make a fly')

    # 伊迪丝想知道这个网站是否会记住她的清单
    [...]
```

再次运行功能测试，看重构前后的表现是否一致：

```
AssertionError: '1: Buy peacock feathers' not found in ['1: Use peacock
feathers to make a fly']
```

很好。接下来，提交这次针对功能测试的小重构：

```
$ git diff # 查看functional_tests.py中的改动
$ git commit -a
```

继续开发工作。如果要处理不止一个待办事项，需要某种持久化存储，在 Web 应用领域，数据库是一种成熟的解决方案。

5.5 Django ORM和第一个模型

对象关系映射器（Object-Relational Mapper，ORM）是一个数据抽象层，描述存储在数据库中的表、行和列。处理数据库时，可以使用熟悉的面向对象方式，写出更好的代码。在 ORM 的概念中，类对应数据库中的表，属性对应列，类的单个实例表示数据库中的一行数据。

Django 对 ORM 提供了良好的支持，学习 ORM 的绝佳方法是在单元测试中使用它，因为单元测试能按照指定方式使用 ORM。

下面在 lists/tests.py 文件中新建一个类：

lists/tests.py

```
from lists.models import Item
[...]

class ItemModelTest(TestCase):

    def test_saving_and_retrieving_items(self):
        first_item = Item()
        first_item.text = 'The first (ever) list item'
        first_item.save()

        second_item = Item()
        second_item.text = 'Item the second'
        second_item.save()

        saved_items = Item.objects.all()
        self.assertEqual(saved_items.count(), 2)
```

```
        first_saved_item = saved_items[0]
        second_saved_item = saved_items[1]
        self.assertEqual(first_saved_item.text, 'The first (ever) list item')
        self.assertEqual(second_saved_item.text, 'Item the second')
```

由上述代码可以看出，在数据库中创建新记录的过程很简单：先创建一个对象，再为一些属性赋值，然后调用 `.save()` 函数。Django 提供了一个查询数据库的 API，即类属性 `.objects`。再使用可能是最简单的查询方法 `.all()`，取回这个表中的全部记录。得到的结果是一个类似列表的对象，叫 `QuerySet`。从这个对象中可以提取出单个对象，然后还可以再调用其他函数，例如 `.count()`。接着，检查存储在数据库中的对象，看保存的信息是否正确。

Django 中的 ORM 有很多有用且直观的功能。现在可能是略读 Django 教程（https://docs.djangoproject.com/en/1.11/intro/tutorial01/）的好时机，这个教程很好地介绍了 ORM 的功能。

> 这个单元测试写得很啰唆，因为我想借此介绍 Django ORM。我不建议你在现实中也这么写。第 15 章会重写这个测试，尽可能做到精简。

术语：单元测试和集成测试的区别以及数据库

追求纯粹的人会告诉你，真正的单元测试绝不能涉及数据库操作。我刚编写的测试或许叫作“整合测试”（integrated test）更确切，因为它不仅测试代码，还依赖于外部系统，即数据库。

现在可以忽略这种区别，因为有两种测试，一种是功能测试，从用户的角度出发，站在一定高度上测试应用；另一种从程序员的角度出发，做底层测试。

在第 23 章会再次讨论单元测试和整合测试。

试着运行单元测试。接下来要进入另一次“单元测试 / 编写代码”循环：

```
ImportError: cannot import name 'Item'
```

很好。下面在 lists/models.py 中写入一些代码，让它有内容可导入。我们有自信，因此会跳过编写 `Item = None` 这一步，直接创建类：

lists/models.py

```
from django.db import models

class Item(object):
    pass
```

这些代码让测试向前进展到了：

```
    first_item.save()
AttributeError: 'Item' object has no attribute 'save'
```

为了给 `Item` 类提供 `save` 方法，也为了让这个类变成真正的 Django 模型，要让它继承 `Model` 类：

lists/models.py

```
from django.db import models

class Item(models.Model):
    pass
```

5.5.1 第一个数据库迁移

再次运行测试，会看到一个数据库错误：

```
django.db.utils.OperationalError: no such table: lists_item
```

在 Django 中，ORM 的任务是模型化数据库。创建数据库其实是由另一个系统负责的，叫作**迁移**（migration）。迁移的任务是，根据你对 models.py 文件的改动情况，添加或删除表和列。

你可以把迁移想象成数据库使用的版本控制系统。后面会看到，把应用部署到线上服务器升级数据库时，迁移十分有用。

现在只需要知道如何创建第一个数据库迁移——使用 `makemigrations` 命令创建迁移：[2]

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0001_initial.py
    - Create model Item
$ ls lists/migrations
0001_initial.py __init__.py __pycache__
```

如果好奇，可以看一下迁移文件中的内容，你会发现，这些内容表明了在 models.py 文件中添加的内容。

与此同时，应该会发现测试又取得了一点进展。

5.5.2 测试向前走得挺远

其实，测试向前走得还挺远：

```
$ python manage.py test lists
[...]
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
AttributeError: 'Item' object has no attribute 'text'
```

这离上次失败的位置整整八行。在这八行代码中，保存了两个待办事项，检查它们是否存入了数据库。可是，Django 似乎不记得有 `.text` 属性。

如果你刚接触 Python，可能会觉得意外，为什么一开始能给 `.text` 属性赋值呢？在 Java 等语言中，本应该得到一个编译错误。但是 Python 要宽松一些。

继承 `models.Model` 的类映射到数据库中的一个表。默认情况下，这种类会得到一个自动生成的 `id` 属性，作为表的主键，但是其他列都要自行定义。定义文本字段的方法如下：

注 2：你可能想知道什么时候应该运行迁移，什么时候应该创建迁移。别急，后面会讲到。

lists/models.py

```
class Item(models.Model):
    text = models.TextField()
```

Django 提供了很多其他字段类型，例如 `IntegerField`、`CharField`、`DateField` 等。使用 `TextField` 而不用 `CharField`，是因为后者需要限制长度，但是就目前而言，这个字段的长度是随意的。关于字段类型的更多介绍可以阅读 Django 教程（https://docs.djangoproject.com/en/1.11/intro/tutorial02/#creating-models）和文档（https://docs.djangoproject.com/en/1.11/ref/models/fields/）。

5.5.3　添加新字段就要创建新迁移

运行测试，会看到另一个数据库错误：

```
django.db.utils.OperationalError: no such column: lists_item.text
```

出现这个错误的原因是在数据库中添加了一个新字段，所以要再创建一个迁移。测试能告诉我们这一点真是太好了！

创建迁移试试：

```
$ python manage.py makemigrations
You are trying to add a non-nullable field 'text' to item without a default; we
can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null
value for this column)
 2) Quit, and let me add a default in models.py
Select an option:2
```

这个命令不允许添加没有默认值的列。选择第二个选项，然后在 models.py 中设定一个默认值。我想你会发现所用的句法无须过多解释：

lists/models.py

```
class Item(models.Model):
    text = models.TextField(default='')
```

现在应该可以顺利创建迁移了：

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0002_item_text.py
    - Add field text to item
```

在 models.py 中添加了两行新代码，创建了两个数据库迁移，由此得到的结果是，模型对象上的 `.text` 属性能被识别为一个特殊属性了，因此属性的值能保存到数据库中，测试也能通过了：

```
$ python manage.py test lists
[...]

Ran 3 tests in 0.010s
OK
```

下面提交创建的第一个模型：

```
$ git status # 看到tests.py和models.py以及两个没跟踪的迁移文件
$ git diff # 审查tests.py和models.py中的改动
$ git add lists
$ git commit -m "Model for list Items and associated migration"
```

5.6 把POST请求中的数据存入数据库

接下来，要修改针对首页中 POST 请求的测试。希望视图把新添加的待办事项存入数据库，而不是直接传给响应。为了测试这个操作，要在现有的测试方法 test_can_save_a_POST_request 中添加三行新代码：

lists/tests.py

```
def test_can_save_a_POST_request(self):
    response = self.client.post('/', data={'item_text': 'A new list item'})

    self.assertEqual(Item.objects.count(), 1)  ❶
    new_item = Item.objects.first()  ❷
    self.assertEqual(new_item.text, 'A new list item')  ❸

    self.assertIn('A new list item', response.content.decode())
    self.assertTemplateUsed(response, 'home.html')
```


❶ 检查是否把一个新 Item 对象存入数据库。objects.count() 是 objects.all().count() 的简写形式。

❷ objects.first() 等价于 objects.all()[0]。

❸ 检查待办事项的文本是否正确。

这个测试变得有点儿长，看起来要测试很多不同的东西。这也是一种**代码异味**。长的单元测试可以分解成两个，或者表明测试太复杂。我们把这个问题记在自己的待办事项清单中，或许可以写在一张便签[3]上：



记在便签上就不会忘记。然后在合适的时候再回来解决。再次运行测试，会看到一个预期失败：

注 3：这张便签由几张图片合成。——译者注

```
        self.assertEqual(Item.objects.count(), 1)
    AssertionError: 0 != 1
```

修改一下视图：

lists/views.py

```
from django.shortcuts import render
from lists.models import Item

def home_page(request):
    item = Item()
    item.text = request.POST.get('item_text', '')
    item.save()

    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', ''),
    })
```

我使用的方法很天真，你或许能发现有一个明显的问题：每次请求首页都保存一个无内容的待办事项。把这个问题记在便签上，稍后再解决。要知道，除了这个明显的严重问题之外，目前还无法为不同的用户创建不同的清单。暂且忽略这些问题。

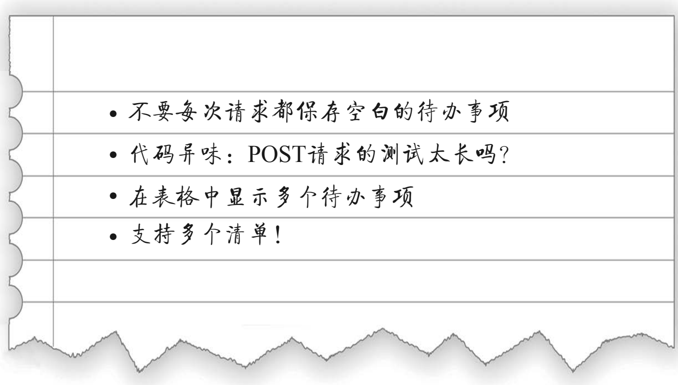
记住，并不是说在实际的开发中始终要把这种明显的问题忽略。预见到问题时，要做出判断，是停止正在做的事从头再来，还是暂时不管，以后再解决。有时完成手头的工作是可以接受的做法，但有些时候问题可能很严重，必须停下来重新思考。

看一下单元测试的进展如何……通过了，太好了！现在可以做些重构：

lists/views.py

```
    return render(request, 'home.html', {
        'new_item_text': item.text
    })
```

看一下便签，我添加了好几件事：



先看第一个问题。虽然可以在现有的测试中添加一个断言，但最好让单元测试一次只测试一件事。那么，定义一个新测试方法吧：

lists/tests.py

```
class HomePageTest(TestCase):
    [...]

    def test_only_saves_items_when_necessary(self):
        self.client.get('/')
        self.assertEqual(Item.objects.count(), 0)
```

这个测试得到的是 `1 != 0` 失败。下面来修正这个问题。注意，虽然对视图函数的逻辑改动幅度很小，但代码的实现方式有很多细微的变动：

lists/views.py

```
def home_page(request):
    if request.method == 'POST':
        new_item_text = request.POST['item_text']   ❶
        Item.objects.create(text=new_item_text)   ❷
    else:
        new_item_text = ''   ❶

    return render(request, 'home.html', {
        'new_item_text': new_item_text,   ❶
    })
```

❶ 使用一个名为 `new_item_text` 的变量，其值是 POST 请求中的数据，或者是空字符串。

❷ `.objects.create` 是创建新 `Item` 对象的简化方式，无须再调用 `.save()` 方法。

这样修改之后，测试就通过了：

```
Ran 4 tests in 0.010s

OK
```

5.7　处理完POST请求后重定向

可是 `new_item_text = ''` 还是让我高兴不起来。幸好现在可以顺带解决这个问题。视图函数有两个作用：一是处理用户输入，二是返回适当的响应。前者已经完成了，即把用户的输入保存到数据库中。下面来看后者。

人们都说处理完 POST 请求后一定要重定向，那么接下来就实现这个功能吧。再次修改针对保存 POST 请求数据的单元测试，不让它渲染包含待办事项的响应，而是重定向到首页：

lists/tests.py

```
    def test_can_save_a_POST_request(self):
        response = self.client.post('/', data={'item_text': 'A new list item'})
```

```
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')

        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/')
```

不需要再拿响应中的 `.content` 属性值和渲染模板得到的结果比较，因此把相应的断言删掉了。现在，响应是 HTTP **重定向**，状态码是 302，让浏览器指向一个新地址。

修改之后运行测试，得到的结果是 `200 != 302` 错误。现在可以大幅度清理视图函数了：

lists/views.py(ch05l028)

```
from django.shortcuts import redirect, render
from lists.models import Item

def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/')

    return render(request, 'home.html')
```

现在，测试应该可以通过了：

```
Ran 4 tests in 0.010s

OK
```

更好的单元测试实践方法：一个测试只测试一件事

现在视图函数处理完 POST 请求后会重定向，这是习惯做法，而且单元测试也一定程度上缩短了，不过还可以做得更好。

良好的单元测试实践方法要求，一个测试只能测试一件事。因为这样便于查找问题。如果一个测试中有多个断言，一旦前面的断言导致测试失败，就无法得知后面的断言情况如何。下一章会看到，如果不小心破坏了视图函数，我们想知道到底是保存对象时出错了，还是响应的类型不对。

刚开始可能无法写出只有一个断言的完美单元测试，不过现在似乎是把正在开发的功能分开测试的好机会：

lists/tests.py

```
    def test_can_save_a_POST_request(self):
        self.client.post('/', data={'item_text': 'A new list item'})

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')
```

```
    def test_redirects_after_POST(self):
        response = self.client.post('/', data={'item_text': 'A new list item'})
        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/')
```

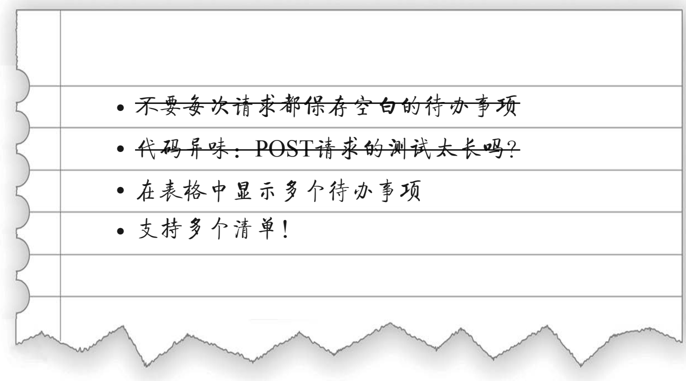
现在应该看到有五个测试通过，而不是四个：

```
Ran 5 tests in 0.010s

OK
```

5.8 在模板中渲染待办事项

感觉好多了！再看待办事项清单。



把问题从清单上划掉几乎和看着测试通过一样让人满足。

第三个问题是最后一个容易解决的问题。要编写一个新单元测试，检查模板是否也能显示多个待办事项：

lists/tests.py

```
class HomePageTest(TestCase):
    [...]

    def test_displays_all_list_items(self):
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        response = self.client.get('/')

        self.assertIn('itemey 1', response.content.decode())
        self.assertIn('itemey 2', response.content.decode())
```

看到测试中的空行了吗？我把设置测试的开头两行放在一起，中间放一行，调用要测试的代码，最后再下断言。这不是强制要求，但是有助于分清测试的结构。设置 – 使用 – 断言，这是单元测试的典型结构。

这个测试和预期一样会失败：

```
AssertionError: 'itemey 1' not found in '<html>\n    <head>\n [...]
```

Django 的模板句法中有一个用于遍历列表的标签，即 {% for .. in .. %}。可以按照下面的方式使用这个标签：

lists/templates/home.html

```
<table id="id_list_table">
    {% for item in items %}
        <tr><td>1: {{ item.text }}</td></tr>
    {% endfor %}
</table>
```

这是模板系统的主要优势之一。现在模板会渲染多个 <tr> 行，每一行对应 items 变量中的一个元素。这么写很优雅！后文我还会介绍更多 Django 模板的魔力，但总有一天你要阅读 Django 文档，学习模板的其他用法。

只修改模板还不能让测试通过，还要在首页的视图中把待办事项传入模板：

lists/views.py

```
def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/')

    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

这样单元测试就能通过了。关键时刻到了，功能测试能通过吗？

```
$ python functional_tests.py
[...]
AssertionError: 'To-Do' not found in 'OperationalError at /'
```

很显然不能。要使用另一种功能测试调试技术，也是最直观的一种：手动访问网站。在浏览器中打开 http://localhost:8000，你会看到一个 Django 调试页面，提示“no such table: lists_item”（没有这个表：lists_item），如图 5-2 所示。

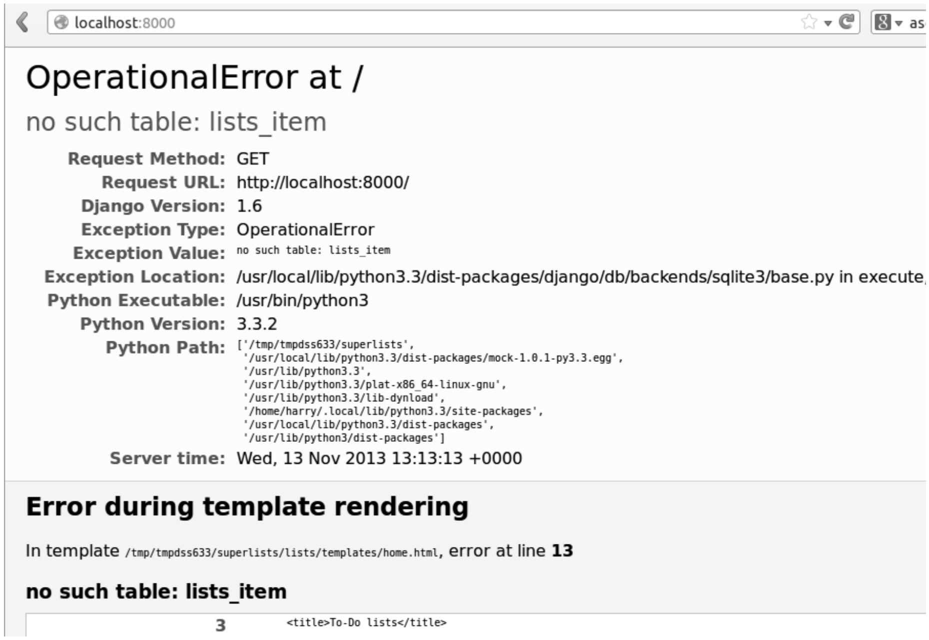


图 5-2：又一个很有帮助的调试信息

5.9 使用迁移创建生产数据库

又是一个 Django 生成的很有帮助的错误消息，大意是说没有正确设置数据库。你可能会问："为什么在单元测试中一切都运行良好呢？"这是因为 Django 为单元测试创建了专用的**测试数据库**——这是 Django 中 `TestCase` 所做的神奇事情之一。

为了设置好真正的数据库，要创建一个数据库。SQLite 数据库只是硬盘中的一个文件。你会在 Django 的 settings.py 文件中发现，默认情况下，Django 把数据库保存为 db.sqlite3，放在项目的基目录中：

superlists/settings.py

```
[...]
# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

我们已经在 models.py 文件和后来创建的迁移文件中告诉 Django 创建数据库所需的一切信息，为了创建真正的数据库，要使用 Django 中另一个强大的 manage.py 命令——`migrate`：

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, lists, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying lists.0001_initial... OK
  Applying lists.0002_item_text... OK
  Applying sessions.0001_initial... OK
```

现在，可以刷新 localhost 上的页面了，你会发现错误页面不见了 [4]。然后再运行功能测试试试：

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers', '1: Use peacock feathers to make a fly']
```

快成功了，只需要让清单显示正确的序号即可。另一个出色的 Django 模板标签 `forloop.counter` 能帮忙解决这个问题：

lists/templates/home.html

```
    {% for item in items %}
        <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
    {% endfor %}
```

再试一次，应该会看到功能测试运行到最后了：

```
    self.fail('Finish the test!')
AssertionError: Finish the test!
```

不过运行测试时，你可能会注意到有什么地方不对劲，如图 5-3 所示。

注 4：如果你看到了另一个错误页面，重启开发服务器试试。Django 可能被发生在眼皮子底下的事情搞糊涂了。

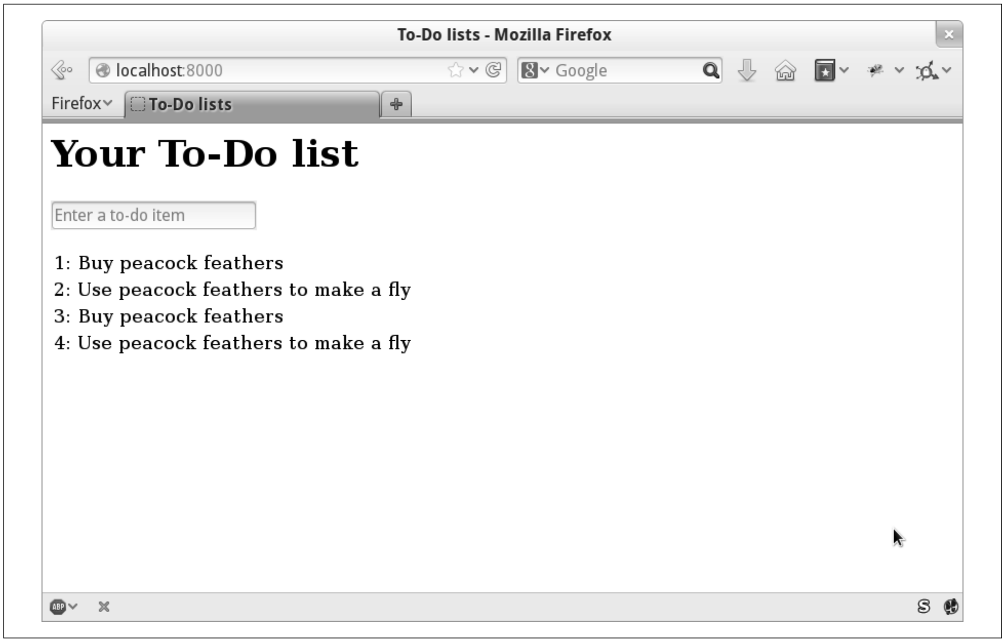


图 5-3：有上一次运行测试时遗留下来的待办事项

哦，天呐。看起来上一次运行测试时在数据库中遗留了数据。如果再次运行测试，会发现待办事项又多了：

```
1: Buy peacock feathers
2: Use peacock feathers to make a fly
3: Buy peacock feathers
4: Use peacock feathers to make a fly
5: Buy peacock feathers
6: Use peacock feathers to make a fly
```

啊，离成功就差一点点了。需要一种自动清理机制。你可以手动清理，方法是先删除数据库再执行 `migrate` 命令新建：

```
$ rm db.sqlite3
$ python manage.py migrate --noinput
```

清理之后要确保功能测试仍能通过。

除了功能测试中这个小问题之外，我们的代码基本上都可以正常运行了。下面做一次提交吧。

先执行 **`git status`**，再执行 **`git diff`**，你应该会看到对 home.html、tests.py 和 views.py 所做的改动。然后提交这些改动：

```
$ git add lists
$ git commit -m "Redirect after POST, and show all items in template"
```

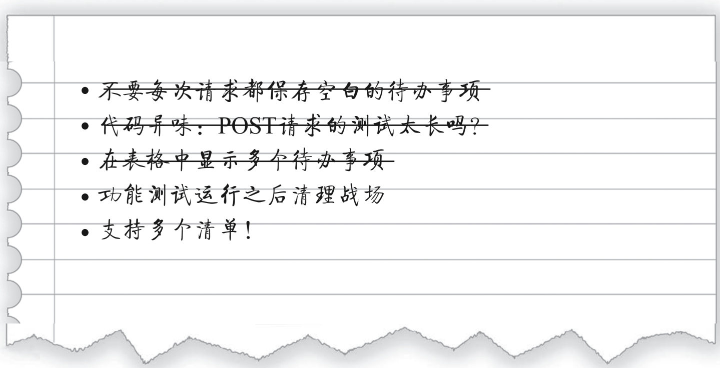
你可能会觉得在每一章结束时做个标记很有用，例如在本章结束时可以这么做：**`git tag end-of-chapter-05`**。

5.10 回顾

这一章我们做了什么呢？

- 编写了一个表单，使用 POST 请求把新待办事项添加到清单中。
- 创建了一个简单的数据库模型，用来存储待办事项。
- 学习了如何创建数据库迁移，既针对测试数据库（自动运行），也针对真实的数据库（手动运行）。
- 用到了两个 Django 模板标签：`{% csrf_token %}` 和 `{% for ... endfor %}` 循环。
- 至少用到了三种功能测试调试技术：行间 `print` 语句、`time.sleep` 以及改进错误消息。

但待办事项清单中还有两件事没做，其中一项是“功能测试运行完毕后清理战场”，还有一项或许更紧急——“支持多个清单”。



我想说的是，虽然网站现在这个样子**可以**发布，但用户可能会觉得奇怪：为什么所有人要共用一个待办事项清单。我想这会让人们停下来思考一些问题：我们彼此之间有怎样的联系？在地球这艘宇宙飞船上有着怎样的共同命运？要如何团结起来解决共同面对的全球性问题？

可实际上，这个网站还没什么用。

先这样吧。

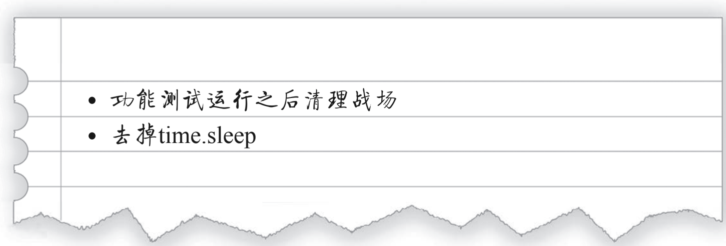
有用的 TDD 概念

- 回归
  新添加的代码破坏了应用原本可以正常使用的功能。
- 意外失败
  测试在意料之外失败了。这意味着测试中有错误，或者测试帮我们发现了一个回归，因此要在代码中修正。
- 遇红 / 变绿 / 重构
  描述 TDD 流程的另一种方式。先编写一个测试看着它失败（遇红），然后编写代码让测试通过（变绿），最后重构，改进实现方式。
- 三角法
  添加一个测试，专门为某些现有的代码编写用例，以此推断出普适的实现方式（在此之前的实现方式可能作弊了）。
- 事不过三，三则重构
  判断何时删除重复代码时使用的经验法则。如果两段代码很相似，往往还要等到第三段相似代码出现，才能确定重构时哪一部分是真正共通、可重用的。
- 记在便签上的待办事项清单
  在便签上记录编写代码过程中遇到的问题，等手头的工作完成后再回过头来解决。

第 6 章

改进功能测试：确保隔离，去掉含糊的休眠

在深入分析和解决真正的问题之前，先来做些清理工作。前一章末尾指出，两次运行的测试之间会彼此影响，那就来修正这个问题吧。此外，我对代码中多次出现的 `time.sleep` 也不满意；这样做似乎有点不科学，我们将采用更可靠的方式实现。



这两项改动都向着测试“最佳实践”迈进，能让测试更确定、更可靠。

6.1　确保功能测试之间相互隔离

前一章结束时留下了一个典型的测试问题：如何**隔离**测试。运行功能测试后待办事项一直存在于数据库中，这会影响下次测试的结果。

运行**单元**测试时，Django 的测试运行程序会自动创建一个全新的测试数据库（和应用真正使用的数据库不同），运行每个测试之前都会清空数据库，等所有测试都运行完之后，再删除这个数据库。但是功能测试目前使用的是应用真正使用的数据库 db.sqlite3。

这个问题的解决方法之一是自己动手，在 functional_tests.py 中添加执行清理任务的代码。这样的任务最适合在 `setUp` 和 `tearDown` 方法中完成。

不过从 1.4 版开始，Django 提供的一个新类 `LiveServerTestCase` 可以代我们完成这一任务。这个类会自动创建一个测试数据库（跟单元测试一样），并启动一个开发服务器，让功能测试在其中运行。虽然这个工具有一定局限性（稍后解决），不过在现阶段十分有用。下面学习如何使用。

`LiveServerTestCase` 必须使用 manage.py，由 Django 的测试运行程序运行。从 Django 1.6 开始，测试运行程序查找所有名字以 test 开头的文件。为了保持文件结构清晰，要新建一个文件夹保存功能测试，让它看起来就像一个应用。Django 对这个文件夹的要求只有一个——必须是有效的 Python 模块，即文件夹中要有一个 __init__.py 文件。

```
$ mkdir functional_tests
$ touch functional_tests/__init__.py
```

然后要**移动**功能测试，把独立的 functional_tests.py 文件移到 `functional_tests` 应用中，并把它重命名为 tests.py。使用 **`git mv`** 命令完成这个操作，让 Git 知道文件移动了：

```
$ git mv functional_tests.py functional_tests/tests.py
$ git status # 显示文件重命名为functional_tests/tests.py，而且新增了__init__.py
```

现在的目录结果如下所示：

```
.
├── db.sqlite3
├── functional_tests
│   ├── __init__.py
│   └── tests.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_item_text.py
│   │   ├── __init__.py
│   │   └── __pycache__
│   ├── models.py
│   ├── __pycache__
│   ├── templates
│   │   └── home.html
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

functional_tests.py 不见了，变成了 functional_tests/tests.py。现在，运行功能测试不执行 `python functional_tests.py` 命令，而是使用 `python manage.py test functional_tests` 命令。

功能测试可以和 `lists` 应用测试混在一起，不过我喜欢把两种测试分开，因为功能测试检测的功能往往存在于不同应用中。功能测试以用户的视角看待事物，而用户并不关心你如何把网站分成不同的应用。

接下来编辑 functional_tests/tests.py，修改 `NewVisitorTest` 类，让它使用 `LiveServerTestCase`：

functional_tests/tests.py (ch06l001)

```
from django.test import LiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time


class NewVisitorTest(LiveServerTestCase):

    def setUp(self):
        [...]
```

继续往下修改。访问网站时，不用硬编码的本地地址（localhost:8000），可以使用 `LiveServerTestCase` 提供的 `live_server_url` 属性：

functional_tests/tests.py (ch06l002)

```
    def test_can_start_a_list_and_retrieve_it_later(self):
        # 伊迪丝听说有一个很酷的在线待办事项应用
        # 她去看了这个应用的首页
        self.browser.get(self.live_server_url)
```

还可以删除文件末尾的 `if __name__ == '__main__'` 代码块，因为之后都使用 Django 的测试运行程序运行功能测试。

现在能使用 Django 的测试运行程序运行功能测试了，指明只运行 `functional_tests` 应用中的测试：

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 65, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!

 ---------------------------------------------------------------------
Ran 1 test in 6.578s

FAILED (failures=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

功能测试和重构前一样，能运行到 self.fail。如果再次运行测试，你会发现，之前的测试不再遗留待办事项了，因为功能测试运行完之后把它们清理掉了。成功了，现在应该提交这次小改动：

```
$ git status # 重命名并修改了functional_tests.py，新增了__init__.py
$ git add functional_tests
$ git diff --staged -M
$ git commit # 提交消息举例："make functional_tests an app, use LiveServerTestCase"
```

git diff 命令中的 -M 标志很有用，意思是“检测移动”，所以 git 会注意到 functional_tests.py 和 functional_tests/tests.py 是同一个文件，显示更合理的差异（去掉这个旗标试试）。

只运行单元测试

现在，如果执行 manage.py test 命令，Django 会运行功能测试和单元测试：

```
$ python manage.py test
Creating test database for alias 'default'...
......F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
[...]
AssertionError: Finish the test!

 ----------------------------------------------------------------------
Ran 7 tests in 6.732s

FAILED (failures=1)
```

如果只想运行单元测试，可以指定只运行 lists 应用中的测试：

```
$ python manage.py test lists
Creating test database for alias 'default'...
......
 ----------------------------------------------------------------------
Ran 6 tests in 0.009s

OK
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

有用的命令（更新版）

- 运行功能测试

  python manage.py test functional_tests

- 运行单元测试

  python manage.py test lists

如果我说“运行测试”，而你不确定我指的是哪一种测试怎么办？可以回顾一下第 4 章最后一节中的流程图，试着找出我们处在哪一步。通常只在所有单元测试都通过后才会运行功能测试。如果不清楚，两种测试都运行试试吧！

6.2　升级Selenium和Geckodriver

今天再次检查这一章时，我发现功能测试停在那里不动了。

后来我才发现，是因为 Firefox 在夜里自动更新了，而 Selenium 和 Geckodriver 也要随之升级。Geckodriver 的发布页面也证实确实有新版发布了。所以，我们要按照下述步骤进行下载和升级。

- 执行 `pip install --upgrade selenium` 命令。
- 下载新版 Geckodriver。
- 备份旧版，放在某处，把新版放在 `PATH` 中的某个位置。
- 执行 `geckodriver --version` 命令，确认新版是否可用。

升级之后，功能测试又能按预期运行了。

在此处讲解这个问题没有特殊的原因。当你阅读到这里时也不一定会遇到这个问题，但是你总会遇到的。再加上我们又在做清理工作，所以我觉得现在是最适合的时机。

这是使用 Selenium 时你必须忍受的一件事。虽然可以固定所用的浏览器和 Selenium 版本（例如在 CI 服务器上），但是现实中的浏览器是不断进化的，你要跟上用户的步伐。

> 只要发现功能测试遇到奇怪的问题，就可以升级 Selenium 试试。

回到常规的编程上来。

6.3　隐式等待、显式等待和含糊的`time.sleep`

来看看功能测试中的 `time.sleep`：

functional_tests/tests.py

```
# 她按回车键后，页面更新了
# 待办事项清单中显示了“1: Buy peacock feathers”
inputbox.send_keys(Keys.ENTER)
time.sleep(1)

self.check_for_row_in_list_table('1: Buy peacock feathers')
```

这叫“显式等待”，与之相对的是“隐式等待”：某些情况下，当 Selenium 认为页面正在加载时，它会“自动”等待一会儿。如果要在页面中查找的元素尚未出现，还可以使用 Selenium 提供的 `implicitly_wait` 方法指明要等多久。

其实本书第 1 版完全依赖隐式等待。但是，隐式等待有点奇怪，而且从 Selenium 3 开始，隐式等待变得极度不可靠。此外，Selenium 团队普遍认为隐式等待不是个好主意，应该避免使用。

因此，第 2 版从一开始就使用显式等待。但问题是，`time.sleep` 自身也有问题。我们现在等待了 1 秒，但谁又能说这是合理的时间呢？对于在自己的设备上运行的多数测试来说，1 秒太长了，严重拖慢了功能测试，等待 0.1 秒就行了。但问题是，如果真等待这么短的时间，常常会导致测试假失败——因为在某些情况下，笔记本电脑的速度可能稍慢一些。话说回来，即便等待了 1 秒，也无法绝对避免不是由真正的问题导致的失败。测试中的假阳性确实烦人（关于这一话题的详细讨论参见 Martin Fowler 写的一篇文章，名为“Eradicating Non-Determinism in Tests”）。

> 如果遇到意料之外的 `NoSuchElementException` 和 `StaleElementException` 错误，通常就表明没有显式等待。去掉 `time.sleep` 试试看会不会出现这样的错误。

下面调整休眠的实现方式，让测试等待足够长的时间，以便捕获可能出现的问题。我们将 `check_for_row_in_list_table` 重命名为 `wait_for_row_in_list_table`，并添加一些轮询 / 重试逻辑：

functional_tests/tests.py (ch06l004)

```
from selenium.common.exceptions import WebDriverException

MAX_WAIT = 10  ❶
[...]

    def wait_for_row_in_list_table(self, row_text):
        start_time = time.time()
        while True:  ❷
            try:
                table = self.browser.find_element_by_id('id_list_table')  ❸
                rows = table.find_elements_by_tag_name('tr')
                self.assertIn(row_text, [row.text for row in rows])
                return  ❹
            except (AssertionError, WebDriverException) as e:  ❺
                if time.time() - start_time > MAX_WAIT:  ❻
                    raise e  ❻
                time.sleep(0.5)  ❺
```

❶ 通过 `MAX_WAIT` 常量设定准备等待的最长时间。10 秒应该足够捕获潜在的问题或不可预知的缓慢因素了。

❷ 这个循环一直运行，直到遇到两个出口中的一个为止。

❸ 这个三行断言跟这个方法的旧版一样。

❹ 如果能顺利运行而且断言通过了，就退出函数、跳出循环。

❺ 但如果捕获到了异常，就再等一小段时间，然后重新循环。我们要捕获两种异常：一种是 `WebDriverException`，在页面未加载或 Selenium 未在页面上找到表格元素时抛出；另一种是 `AssertionError`，因为页面中虽有表格，但它可能在页面重新加载之前就存在，里面还是没有我们要找的行。

❻ 这是第二个出口。如果执行到这里，说明代码不断抛出异常，已经超时。因此这里再次抛出异常，向上冒泡，最终可能出现在调用跟踪中，指明测试失败的原因。

是不是觉得这段代码有点蹩脚、有点不明所以？我同意。后文会做重构，定义一个通用的 `wait_for` 辅助方法，把计时、重新抛出异常的逻辑与测试断言分开。不过，这要等到需要在多个地方使用它时再做。

> 如果你以前用过 Selenium，可能知道它提供了一些用于等待的辅助函数。我不太喜欢使用这些函数。本书将构建几个用于等待的辅助工具，我觉得这样能让代码更优雅、更易于阅读。不过，你绝对应该学习一下 Selenium 自带的那些辅助函数，然后判断要不要使用。

下面调用新的方法，并把含糊的 `time.sleep` 去掉：

functional_tests/tests.py (ch06l005)

```
        [...]
        # 她按回车键后，页面更新了
        # 待办事项清单中显示了“1: Buy peacock feathers”
        inputbox.send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table('1: Buy peacock feathers')


        # 页面中还有一个文本框，可以输入其他的待办事项
        # 她输入了“Use peacock feathers to make a fly”（使用孔雀羽毛做假蝇）
        # 伊迪丝做事很有条理
        inputbox = self.browser.find_element_by_id('id_new_item')
        inputbox.send_keys('Use peacock feathers to make a fly')
        inputbox.send_keys(Keys.ENTER)

        # 页面再次更新，清单中显示了这两个待办事项
        self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
        self.wait_for_row_in_list_table('1: Buy peacock feathers')
        [...]
```

然后再次运行测试：

```
$ python manage.py test
Creating test database for alias 'default'...
......F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 73, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!

 ---------------------------------------------------------------------
```

```
Ran 7 tests in 4.552s

FAILED (failures=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

结果跟之前一样，不过测试的执行时间比之前短了几秒。虽然现在只快了一点，但是随着测试的增多，速度优势便会慢慢显现。

为了确认我们做得对不对，下面将故意破坏测试，看看会出现什么错误。首先看一下能否检测永远不会出现在一行里的文本：

functional_tests/tests.py (ch06l006)

```
            rows = table.find_elements_by_tag_name('tr')
            self.assertIn('foo', [row.text for row in rows])
            return
```

我们将看到一个意思明确的测试失败消息：

```
    self.assertIn('foo', [row.text for row in rows])
AssertionError: 'foo' not found in ['1: Buy peacock feathers']
```

改回去，然后再破坏其他地方：

functional_tests/tests.py (ch06l007)

```
        try:
            table = self.browser.find_element_by_id('id_nothing')
            rows = table.find_elements_by_tag_name('tr')
            self.assertIn(row_text, [row.text for row in rows])
            return
        [...]
```

显然，得到的错误指明页面中没有要查找的元素：

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_nothing"]
```

看起来一切正常。改回原样，最后再运行测试确认一下：

```
$ python manage.py test
[...]
AssertionError: Finish the test!
```

很好。短暂离题之后，下面回到正轨，说明如何实现多个清单。

本章运用的测试“最佳实践”

- 确保测试隔离，管理全局状态

  不同的测试之间不能彼此影响，也就是说每次测试结束后都要还原永久状态。Django 的测试运行程序可以帮助我们创建测试数据库，这个数据库在每次测试结束后都会清空（详情请参见第 23 章）。

- 避免使用“含糊的”休眠

  一旦需要等待什么加载，我们的第一反应便是使用 `time.sleep`。但是这样做带来的问题是，时间的长度是两眼一抹黑：要么太短，容易导致假失败；要么太长，会拖慢测试。我推荐使用重试循环，它可以轮询应用，尽早向前行进。

- 不要依赖Selenium的隐式等待

  Selenium 确实有理论上的“隐式”等待，但是在不同浏览器上的实现各不相同。而且在写作本书时，隐式等待在 Selenium 3 的 Firefox 驱动上极度不可靠。Python 之禅说道：“明了胜于晦涩”，因此首选显式等待。

第 7 章

步步为营

现在就来解决当前存在的问题。目前，我们的设计只允许创建一个全局清单。这一章将说明一个关键的 TDD 技术：如何使用递增的步进式方法修改现有代码，而且保证代码在修改前后都能正常运行。我们要做测试山羊，而不是重构猫。

7.1　必要时做少量的设计

请想一想应该如何支持多个清单。目前的功能测试（与设计文档最接近）是这样的：

functional_tests/tests.py

```
# 伊迪丝想知道这个网站是否会记住她的清单
# 她看到网站为她生成了一个唯一的URL
# 页面中有一些解说这个功能的文字
self.fail('Finish the test!')

# 她访问那个URL，发现待办事项清单还在

# 她很满意，去睡觉了
```

不过，我们要在此基础上扩展一下，让用户不能相互查看各自的清单，而且每个用户都有自己的 URL，能访问自己的清单。怎么实现这样的设计呢？

7.1.1　不要预先做大量设计

TDD 和软件开发中的敏捷运动联系紧密。敏捷运动反对传统软件工程实践中**预先做大量设计**的做法，因为除了要花费大量时间收集需求之外，设计阶段还要用等量的时间在纸上规划软件。敏捷理念则认为，在实践中解决问题比理论分析能学到更多，而且让应用尽早接受真实用户的检验效果更好。无须花这么多时间提前设计，而要尽早把**最简可用的应用**放

出来，根据实际使用中得到的反馈逐步向前推进设计。

这并不是说要完全禁止思考设计。前一章我们看到，不经思考呆头呆脑往前走，最终也能找到正确答案，不过稍微思考一下设计往往能帮助我们更快地找到答案。那么，下面分析一下这个最简可用的应用，想想应该使用哪种设计方式。

- 想让每个用户都能保存自己的清单，目前来说，至少能保存一个清单。
- 清单由多个待办事项组成，待办事项的主要属性应该是一些描述性文字。
- 要保存清单，以便多次访问。现在，可以为用户提供一个唯一的 URL，指向他们的清单。以后或许需要一种自动识别用户的机制，然后把他们的清单显示出来。

为了实现第一条，看样子要把清单和其中的待办事项存入数据库。每个清单都有一个唯一的 URL，而且清单中的每个待办事项都是一些描述性文字，和所在的清单关联。

7.1.2　YAGNI

关于设计的思考一旦开始就很难停下来，我们会冒出各种想法：或许想给每个清单起个名字或加个标题，或许想使用用户名和密码识别用户，或许想给清单添加一个较长的备注和简短的描述，或许想存储某种顺序，等等。但是，要遵守敏捷理念的另一个信条："YAGNI"（读作 yag-knee）。它是"You ain't gonna need it"（你不需要这个）的简称。作为软件开发者，我们从创造事物中获得乐趣。有时我们冒出一个想法，觉得**可能**需要，便无法抵御内心的冲动想要开发出来。可问题是，不管想法有多好，大多数情况下最终你都**用不到**这个功能。应用中会残留很多没用的代码，还增加了应用的复杂度。YAGNI 是个真言，可以用来抵御热切的创造欲。

7.1.3　REST（式）

我们已经知道怎么处理数据结构，即使用"模型 – 视图 – 控制器"中的模型部分。那视图和控制器部分怎么办呢？在 Web 浏览器中用户怎么处理清单和待办事项呢？

"表现层状态转化"（representational state transfer，REST）是 Web 设计的一种方式，经常用来引导基于 Web 的 API 设计。设计面向用户的网站时，不必**严格**遵守 REST 规则，可是从中能得到一些启发。（如果想看看真实的 REST API 是什么样子，可以跳到附录 F。）

REST 建议 URL 结构匹配数据结构，即这个应用中的清单和其中的待办事项。清单有各自的 URL：

```
/lists/<list identifier>/
```

这个 URL 满足了功能测试中提出的需求。若想查看某个清单，我们可以发送一个 GET 请求（就是在普通的浏览器中访问这个页面）。

若想创建全新的清单，可以向一个特殊的 URL 发送 POST 请求：

```
/lists/new
```

若想在现有的清单中添加一个新待办事项，我们可以向另外一个 URL 发送 POST 请求：

```
/lists/<list identifier>/add_item
```

（再次说明，我们不会严格遵守 REST 规则，只是从中得到启发。比如，按照 REST 规则，

这里应该使用 PUT 请求，但是标准的 HTML 表单无法发送 PUT 请求。）

概括起来，本章的便签如下所示：



7.2 使用TDD实现新设计

应该如何使用 TDD 实现新的设计呢？再回顾一下 TDD 的流程图，如图 7-1 所示。

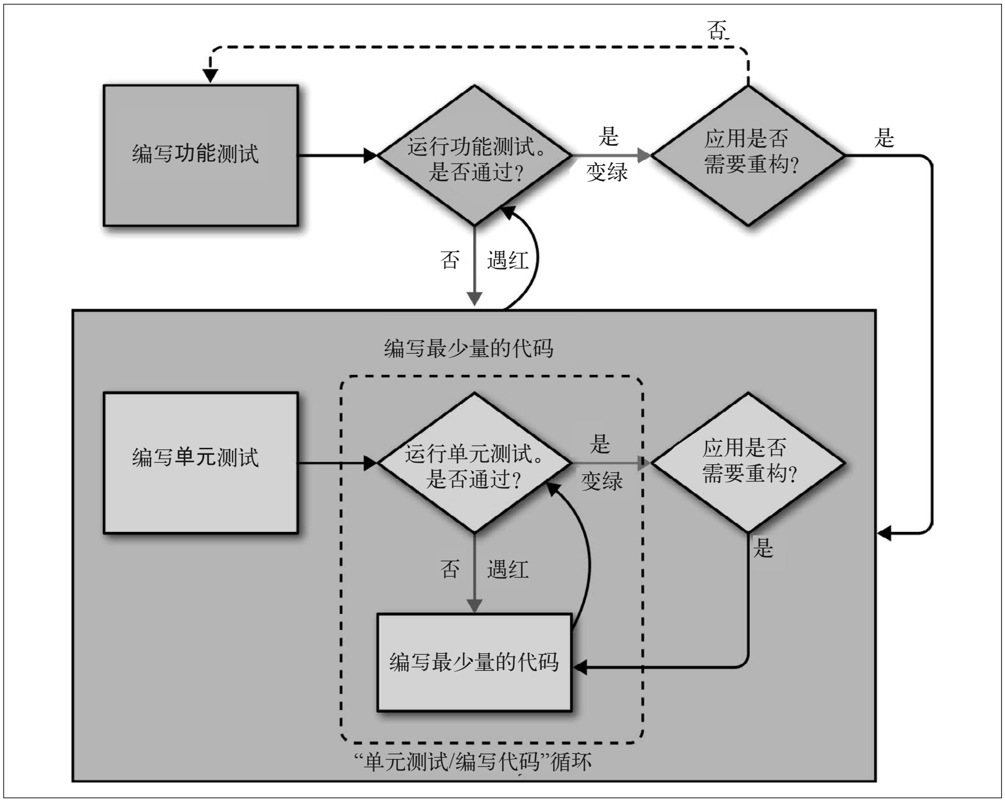


图 7-1：包含功能测试和单元测试的 TDD 流程

在流程的外层，既要添加新功能（添加新的功能测试，再编写新的应用代码），也要重构应用的代码，即重写部分现有的实现，保持应用的功能不变，但使用新的设计方式。我们通过现有的功能测试确保不破坏现有的功能，同时通过新功能测试驱动开发新功能。

在单元测试层，要添加新测试或者修改现有的测试以检查想改动的功能，没改动的测试则用来保证这个过程没有破坏现有的功能。

7.3　确保出现回归测试

下面根据便签上的待办事项编写一个新的功能测试方法，引入第二个用户，并确认他的待办事项清单与伊迪丝的清单是分开的。

这个功能测试的开头与第一个功能测试基本一样——伊迪丝提交第一个待办事项后，应用创建一个新清单。不过这次要多添加一个断言，确认伊迪丝的清单是独立的，有唯一的 URL：

functional_tests/tests.py (ch07l005)

```
    def test_can_start_a_list_for_one_user(self):
        # 伊迪丝听说有一个很酷的在线待办事项应用
        # 她去看了这个应用的首页
        [...]
        # 页面再次更新，她的清单中显示了这两个待办事项
        self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
        self.wait_for_row_in_list_table('1: Buy peacock feathers')

        # 她很满意，然后去睡觉了


    def test_multiple_users_can_start_lists_at_different_urls(self):
        # 伊迪丝新建一个待办事项清单
        self.browser.get(self.live_server_url)
        inputbox = self.browser.find_element_by_id('id_new_item')
        inputbox.send_keys('Buy peacock feathers')
        inputbox.send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table('1: Buy peacock feathers')

        # 她注意到清单有个唯一的URL
        edith_list_url = self.browser.current_url
        self.assertRegex(edith_list_url, '/lists/.+')  ❶
```

❶ `assertRegex` 是 `unittest` 提供的辅助函数，用于检查字符串是否匹配正则表达式。我们利用它检查新的 REST 式设计能否实现。详情请参见 `unittest` 的文档。

然后，我们假设有一个新用户正在访问网站。当新用户访问首页时，要测试他不能看到伊迪丝的待办事项，而且他的清单有自己的唯一 URL：

functional_tests/tests.py (ch07l006)

```
        [...]
        self.assertRegex(edith_list_url, '/lists/.+')
```

```
# 现在一名叫作弗朗西斯的新用户访问了网站

## 我们使用一个新浏览器会话  ❶
## 确保伊迪丝的信息不会从cookie中泄露出去
self.browser.quit()
self.browser = webdriver.Firefox()

# 弗朗西斯访问首页
# 页面中看不到伊迪丝的清单
self.browser.get(self.live_server_url)
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Buy peacock feathers', page_text)
self.assertNotIn('make a fly', page_text)

# 弗朗西斯输入一个新待办事项，新建一个清单
# 他不像伊迪丝那样兴趣盎然
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Buy milk')
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')

# 弗朗西斯获得了他的唯一URL
francis_list_url = self.browser.current_url
self.assertRegex(francis_list_url, '/lists/.+')
self.assertNotEqual(francis_list_url, edith_list_url)

# 这个页面还是没有伊迪丝的清单
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Buy peacock feathers', page_text)
self.assertIn('Buy milk', page_text)

# 两人都很满意，然后去睡觉了
```

❶ 按照习惯，我使用两个 # 表示“元注释”。元注释的作用是说明测试的**工作方式**以及为什么这么做。使用两个井号是为了和功能测试中解说用户故事的常规注释区分开。这个元注释是发给未来自己的消息，如果没有这条消息，到时你可能会觉得奇怪，想知道到底为什么要退出浏览器再启动一个新会话。

除了元注释之外，就不需要对这个新测试多做解释了。看一下运行功能测试后的情况如何：

```
$ python manage.py test functional_tests
[...]
.F
======================================================================
FAIL: test_multiple_users_can_start_lists_at_different_urls
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 83, in
test_multiple_users_can_start_lists_at_different_urls
    self.assertRegex(edith_list_url, '/lists/.+')
AssertionError: Regex didn't match: '/lists/.+' not found in
```

```
'http://localhost:8081/'

 ---------------------------------------------------------------------
Ran 2 tests in 5.786s

FAILED (failures=1)
```

很好，第一个测试仍能通过，而第二个测试也如我们所料失败了。先提交一次，然后再编写一些新模型和新视图：

```
$ git commit -a
```

7.4　逐步迭代，实现新设计

我太兴奋了，迫切地想实现新设计，这种欲望太强烈，谁也无法阻拦，我真想现在就开始修改 models.py。但这么做可能会导致一半的单元测试失败，我们不得不一行一行修改代码，而且要一次改完，工作量太大。有这样的冲动很自然，但 TDD 理念一直反对这么做。我们要遵从测试山羊的教诲，不能听信重构猫的谗言。无须一次性实现光鲜亮丽的整个设计，改动的幅度要小一些，每一步都要遵照设计思想的指引，保证修改后应用仍能正常运行。

在待办事项清单中还有四个问题没解决。无法匹配正则表达式的那个功能测试提醒我们，接下来要解决的是第二个问题，即为每个清单添加唯一的 URL 和标识符。下面解决且只解决这个问题。

清单的 URL 出现在重定向 POST 请求之后。在文件 lists/tests.py 中，找到 `test_redirects_after_POST`，修改重定向期望转向的地址：

lists/tests.py

```
self.assertEqual(response.status_code, 302)
self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```

这个 URL 看起来是不是有点儿奇怪？在应用的最终设计中显然不会使用 /lists/the-only-list-in-the-world/ 这个 URL。可是我们承诺过，一次只做一项改动，既然应用现在只支持一个清单，那这就是唯一合理的 URL。我们还在向前进，到时候清单和首页的地址都会变，这是更符合 REST 式设计的一个实现步骤。稍后我们会支持多个清单，也会提供简单的方法修改 URL。

> 我们可以换种想法，把这看成是一种解决问题的技术：新的 URL 设计还没实现，所以这个 URL 可用于没有待办事项的清单。最终要设法解决包含 *n* 个待办事项的清单，不过解决包含一个待办事项的清单是个好的开始。

运行单元测试，会看到一个预期失败：

```
$ python manage.py test lists
[...]
AssertionError: '/' != '/lists/the-only-list-in-the-world/'
```

可以修改文件 lists/views.py 中的 `home_page` 视图：

lists/views.py

```
def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/the-only-list-in-the-world/')

    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

这么修改，功能测试显然会失败，因为网站中并没有这个 URL。毫无疑问，如果运行功能测试，你会看到测试在尝试提交第一个待办事项后失败，提示无法找到显示清单的表格。出现这个错误的原因是，/the-only-list-in-the-world/ 这个 URL 还不存在。

```
  File "/.../superlists/functional_tests/tests.py", line 57, in
test_can_start_a_list_for_one_user
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]

[...]

  File "/.../superlists/functional_tests/tests.py", line 79, in
test_multiple_users_can_start_lists_at_different_urls
    self.wait_for_row_in_list_table('1: Buy peacock feathers')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]
```

不仅新添加的测试失败了，原来那个也失败了。这表明出现了**回归**。接下来就为这个唯一的清单提供一个 URL，重回正常状态。

7.5　自成一体的第一步：新的URL

打开 lists/tests.py，添加一个新测试类，命名为 `ListViewTest`。然后把 `HomePageTest` 类中的 `test_displays_all_list_items` 方法复制到这个新类中。给这个方法重新命名，再做些修改：

lists/tests.py (ch07l009)

```
class ListViewTest(TestCase):

    def test_displays_all_items(self):
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        response = self.client.get('/lists/the-only-list-in-the-world/')

        self.assertContains(response, 'itemey 1')  ❶
        self.assertContains(response, 'itemey 2')  ❶
```

❶ 这里用到一个新的辅助方法：现在不必再使用有点儿烦人的 `assertIn` 和 `response.content.decode()` 了，Django 提供了 `assertContains` 方法，它知道如何处理响应以及响应内容中的字节。

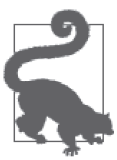
运行这个测试，看看情况：

```
    self.assertContains(response, 'itemey 1')
[...]
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
```

这是使用 `assertContains` 的附加好处——它直接指出测试失败的原因是新 URL 不存在，而且返回的是 404 响应。

7.5.1　一个新URL

现在那个唯一的清单 URL 还不存在，要在 superlists/urls.py 中解决这个问题。



留意 URL 末尾的斜线，在测试中和 urls.py 中都要小心，因为这个斜线往往就是问题的根源。

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/the-only-list-in-the-world/$', views.view_list, name='view_list'),
]
```

再次运行测试，得到的结果如下：

```
AttributeError: module 'lists.views' has no attribute 'view_list'
```

7.5.2　一个新视图函数

这个结果无须过多解说。下面在 lists/views.py 中定义一个新视图函数：

lists/views.py

```
def view_list(request):
    pass
```

现在测试的结果变成了：

```
ValueError: The view lists.views.view_list didn't return an HttpResponse
object. It returned None instead.

[...]
FAILED (errors=1)
```

失败的只有一个了，而且为我们指明了方向。把 `home_page` 视图的最后两行复制过来，看能否骗过测试：

lists/views.py

```
def view_list(request):
    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

再次运行单元测试，测试应该能通过了：

```
Ran 7 tests in 0.016s
OK
```

再运行功能测试，看看情况如何：

```
FAIL: test_can_start_a_list_for_one_user
[...]
  File "/.../superlists/functional_tests/tests.py", line 67, in
test_can_start_a_list_for_one_user
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']

FAIL: test_multiple_users_can_start_lists_at_different_urls
[...]
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
list\n1: Buy peacock feathers'
[...]
```

两个功能测试都有一点进展，不过依然失败了。我们要尽快重回正常状态，让第一个功能测试再次通过。失败消息中有什么线索呢？

可以看出，失败发生在尝试添加第二个待办事项时——看来得调试一番了。我们知道首页是正常的，因为功能测试能执行到第 67 行，也就是至少添加了一个待办事项。而且，单元测试都能通过，因此可以确定 URL 和视图能正常运作——首页使用正确的模板显示、能处理 POST 请求，only-list-in-the-world 视图知道如何显示所有待办事项……但是它不知道怎样处理 POST 请求。啊，这就是线索。

根据经验，第二个线索是，当所有单元测试都能通过而功能测试不能通过时，问题通常是由单元测试没有覆盖的事物引起的——这往往是模板的问题。

最终我们找到了问题的根源：home.html 中的输入表单没有明确指定 POST 的目标 URL。

lists/templates/home.html

```
        <form method="POST">
```

默认情况下，浏览器把 POST 数据发回表单当前所在的 URL。这样的话，在首页能正常运行，但到 only-list-in-the-world 页面就不行了。

找到根源后，我们本可以为新视图添加处理 POST 请求的功能，但是这样还得编写一堆测试和代码，而我们的目的是尽早重回正常状态。其实，修正这个问题最快速的方法是使用现在能正常运行的首页视图处理所有 POST 请求：

lists/templates/home.html

```
        <form method="POST" action="/">
```

再次运行测试，你会发现功能测试回到之前的状态了：

```
FAIL: test_multiple_users_can_start_lists_at_different_urls
[...]
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
list\n1: Buy peacock feathers'

Ran 2 tests in 8.541s
FAILED (failures=1)
```

原先的测试再次通过，由此可以确认我们又回到了正常状态。新功能也许还不可用，但至少旧的功能依旧正常。

7.6　变绿了吗？该重构了

该清理一下测试了。

在**遇红 / 变绿 / 重构**流程中，已经走到“变绿”这一步，接下来该重构了。现在我们有两个视图，一个用于首页，一个用于单个清单。目前，这两个视图共用一个模板，而且传入了数据库中的所有待办事项。如果仔细查看单元测试中的方法，或许会发现某些部分需要修改：

```
$ grep -E "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_uses_home_template(self):
    def test_displays_all_list_items(self):
    def test_can_save_a_POST_request(self):
    def test_redirects_after_POST(self):
    def test_only_saves_items_when_necessary(self):
class ListViewTest(TestCase):
    def test_displays_all_items(self):
class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
```

完全可以把 `HomePageTest` 中的 `test_displays_all_list_items` 方法删除，因为不需要了。如果现在执行 **`manage.py test lists`** 命令，应该会看到运行了 6 个测试，而不是 7 个：

```
Ran 6 tests in 0.016s
OK
```

而且，首页模板其实不用再显示所有的待办事项，而应该只显示一个输入框让用户新建清单。

7.7　再迈一小步：一个新模板，用于查看清单

既然首页和清单视图是不同的页面，它们就应该使用不同的 HTML 模板。home.html 可以只包含一个输入框，新模板 list.html 则在表格中显示现有的待办事项。

下面添加一个新测试，检查是否使用了不同的模板：

lists/tests.py

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        response = self.client.get('/lists/the-only-list-in-the-world/')
        self.assertTemplateUsed(response, 'list.html')


    def test_displays_all_items(self):
        [...]
```

`assertTemplateUsed` 是 Django 测试客户端提供的强大方法之一。看一下测试的结果如何：

```
AssertionError: False is not true : Template 'list.html' was not a template
used to render the response. Actual template(s) used: home.html
```

很好！然后修改视图：

lists/views.py

```
def view_list(request):
    items = Item.objects.all()
    return render(request, 'list.html', {'items': items})
```

不过很显然，这个模板还不存在。如果运行单元测试，会得到如下结果：

```
django.template.exceptions.TemplateDoesNotExist: list.html
```

新建一个文件，保存为 lists/templates/list.html：

```
$ touch lists/templates/list.html
```

这个模板是空的，测试会显示如下错误——幸好有测试，我们才不会忘记输入内容：

```
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```

单个清单的模板会用到目前 home.html 中的很多代码，所以可以先把其中的内容复制过来：

```
$ cp lists/templates/home.html lists/templates/list.html
```

这会让测试再次通过（变绿）。现在要做一些清理工作（重构）。我们说过，首页不用显示待办事项，只放一个新建清单的输入框就行。因此，可以删除 lists/templates/home.html 中的一些代码，或许还可以把 `h1` 改成“Start a new To-Do list”：

lists/templates/home.html

```
<body>
  <h1>Start a new To-Do list</h1>
  <form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
    {% csrf_token %}
  </form>
</body>
```

再次运行测试，确认这次改动没有破坏任何功能。很好，继续清理。

其实也不用在 home_page 视图中把全部待办事项都传入 home.html 模板，因此可以把 home_page 视图简化成：

lists/views.py

```
def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/the-only-list-in-the-world/')
    return render(request, 'home.html')
```

再次运行单元测试，它们仍然能通过。然后运行功能测试：

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
```

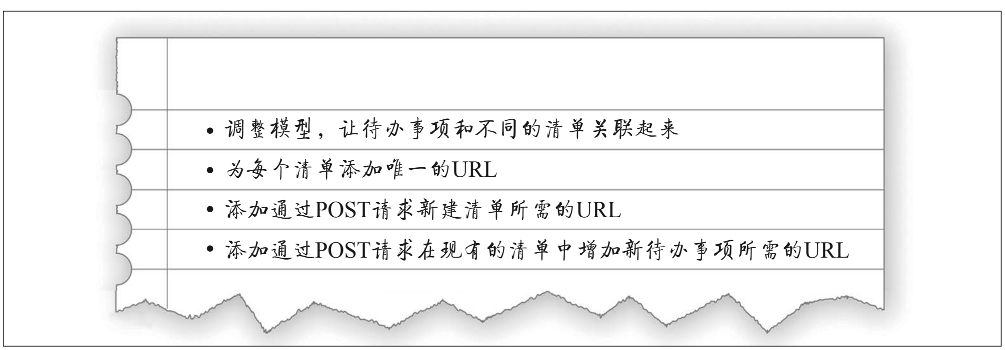
不错，回归测试（第一个功能测试）通过了，而且新增的测试稍微向前进展了一点——可以指出弗朗西斯没有得到自己的清单页面（因为他仍能看到伊迪丝的部分待办事项）。

你可能觉得并没有取得太多进展，因为网站的功能和本章开始时几乎一模一样。其实有进展，我们正在实现新设计，在前进的道路上铺好了几块垫脚石，**而且网站的功能几乎没变**。提交目前取得的进展：

```
$ git status # 会看到4个改动的文件和1个新文件list.html
$ git add lists/templates/list.html
$ git diff # 会看到我们简化了home.html
           # 把一个测试移到了lists/tests.py中的新类里
           # 在views.py中添加了一个新视图
           # 还简化了home_page视图，并在urls.py中增加了一个映射
$ git commit -a # 编写一个消息概述以上操作，或许可以写成
                # “new URL, view and template to display lists”
```

7.8　第三小步：用于添加待办事项的URL

看一下待办事项清单，现在到哪一步了呢？



第二个问题已经取得了**一定**进展，不过网站中还是只有一个清单。第一个问题有点吓人。我们能对第三或第四个问题做些什么呢？

下面添加一个新 URL，用于新建待办事项。这么做至少能简化首页视图。

7.8.1 用来测试新建清单的测试类

打开文件 lists/tests.py，把 `test_can_save_a_POST_request` 和 `test_redirects_after_POST` 两个方法移到一个新类中，然后再修改 POST 请求的目标 URL：

lists/tests.py (ch07l021-1)

```
class NewListTest(TestCase):

    def test_can_save_a_POST_request(self):
        self.client.post('/lists/new', data={'item_text': 'A new list item'})
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')


    def test_redirects_after_POST(self):
        response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```

顺便说一句，这里也要注意末尾的斜线——`/new` 后面不加斜线。我的习惯是，不在修改数据库的“操作”后加斜线。

顺便学习一个新的 Django 测试客户端方法 `assertRedirects`：

lists/tests.py (ch07l021-2)

```
    def test_redirects_after_POST(self):
        response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
        self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
```

这个方法没什么大用，不过能把两个断言精简成一个。

运行这个测试试试：

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
    self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

第一个失败消息告诉我们，新建的待办事项没有存入数据库。第二个失败消息指出视图返回的状态码是 404，而不是表示重定向的 302。这是因为还没把 /lists/new 添加到 URL 映射中，所以 `client.post` 得到的是“not found”（未找到）响应。

还记得之前我们是如何把这种测试分成两个测试方法的吗？如果在一个测试方法中同时测试保存数据和重定向，看到的失败消息就是 `0 != 1`，调试起来更难。如果你好奇我是怎么知道要这么做的，不要犹豫，问我吧。

7.8.2　用于新建清单的URL和视图

下面添加新的 URL 映射：

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/the-only-list-in-the-world/$', views.view_list, name='view_list'),
]
```

再运行测试，会得到 `no attribute'new_list'` 错误。修正这个问题，在文件 lists/views.py 中写入：

lists/views.py (ch07l023-1)

```
def new_list(request):
    pass
```

再运行测试，得到的失败消息是："The view lists.views.new_list didn't return an HttpResponse object"（`lists.views.new_list` 视图没返回 `HttpResponse` 对象）。这个消息很眼熟。虽然可以返回一个原始的 `HttpResponse` 对象，但既然知道需要的是重定向，那就从 `home_page` 视图中借用一行代码吧：

lists/views.py (ch07l023-2)

```
def new_list(request):
    return redirect('/lists/the-only-list-in-the-world/')
```

现在测试的结果是：

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

失败消息简洁易懂。再从 `home_page` 视图中借用一行代码：

lists/views.py (ch07l023-3)

```
def new_list(request):
    Item.objects.create(text=request.POST['item_text'])
    return redirect('/lists/the-only-list-in-the-world/')
```

加入这行代码后，测试便能通过了：

```
Ran 7 tests in 0.030s

OK
```

而且功能测试表明，我们又回到了正常状态：

```
[...]
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
Ran 2 tests in 8.972s
FAILED (failures=1)
```

7.8.3　删除当前多余的代码和测试

看起来不错。既然新视图完成了以前 home_page 视图的大部分工作，应该就可以大幅度精简 home_page 了。 比如说，可以删除整个 if request.method == 'POST' 部分吗？

lists/views.py

```
def home_page(request):
    return render(request, 'home.html')
```

当然可以！

```
OK
```

既然已经动手简化了，还可以把当前多余的测试方法 test_only_saves_items_when_necessary 也删掉。

删掉之后是不是感觉挺好的？视图函数变得更简洁了。再次运行测试，确认一切正常：

```
Ran 6 tests in 0.016s
OK
```

那功能测试呢？

7.8.4　出现回归！让表单指向刚添加的新URL

糟糕：

```
ERROR: test_can_start_a_list_for_one_user
[...]
  File "/.../superlists/functional_tests/tests.py", line 57, in
test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('1: Buy peacock feathers')
  File "/.../superlists/functional_tests/tests.py", line 23, in
wait_for_row_in_list_table
    table = self.browser.find_element_by_id('id_list_table')
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]

ERROR: test_multiple_users_can_start_lists_at_different_urls
[...]
  File "/.../superlists/functional_tests/tests.py", line 79, in
test_multiple_users_can_start_lists_at_different_urls
    self.wait_for_row_in_list_table('1: Buy peacock feathers')
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]
[...]

Ran 2 tests in 11.592s
FAILED (errors=2)
```

这是因为表单依然指向旧的 URL。在 home.html 和 lists.html 中，把表单改成：

lists/templates/home.html, lists/templates/list.html

```
    <form method="POST" action="/lists/new">
```

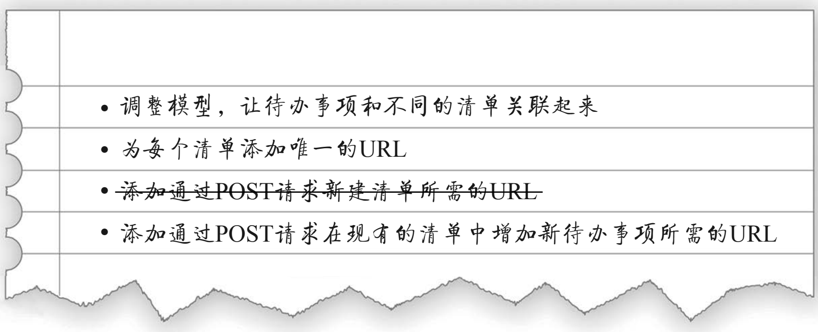
这样就能回到之前的状态了：

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
[...]
FAILED (failures=1)
```

以上操作可以作为一次完整的提交：对 URL 映射做了些改动，views.py 看起来也精简多了，而且能保证应用还能像以前那样正常运行。我们的重构技术变得越来越好了！

```
$ git status # 5个改动的文件
$ git diff # 在两个表单中添加了URL，视图和测试都有代码移动，还添加了一个新URL
$ git commit -a
```

可以在待办事项清单中划掉一个问题了：



7.9　下定决心，调整模型

关于 URL 的清理工作做得够多了，现在下定决心修改模型。先调整模型的单元测试。这次换种方式，以差异的形式表示改动的地方：

lists/tests.py

```
@@ -1,5 +1,5 @@
 from django.test import TestCase
-from lists.models import Item
+from lists.models import Item, List


 class HomePageTest(TestCase):
@@ -44,22 +44,32 @@ class ListViewTest(TestCase):



-class ItemModelTest(TestCase):
+class ListAndItemModelsTest(TestCase):

     def test_saving_and_retrieving_items(self):
```

```
+        list_ = List()
+        list_.save()
+
         first_item = Item()
         first_item.text = 'The first (ever) list item'
+        first_item.list = list_
         first_item.save()

         second_item = Item()
         second_item.text = 'Item the second'
+        second_item.list = list_
         second_item.save()

+        saved_list = List.objects.first()
+        self.assertEqual(saved_list, list_)
+
         saved_items = Item.objects.all()
         self.assertEqual(saved_items.count(), 2)

         first_saved_item = saved_items[0]
         second_saved_item = saved_items[1]
         self.assertEqual(first_saved_item.text, 'The first (ever) list item')
+        self.assertEqual(first_saved_item.list, list_)
         self.assertEqual(second_saved_item.text, 'Item the second')
+        self.assertEqual(second_saved_item.list, list_)
```

新建了一个 `List` 对象，然后通过给 `.list` 属性赋值把两个待办事项归在这个对象名下。要检查这个清单是否正确保存，也要检查是否保存了那两个待办事项与清单之间的关系。你还会注意到可以直接比较两个清单（`saved_list` 和 `list_`）——其实比较的是两个清单的主键（`.id` 属性）是否相同。

> 我使用变量名 `list_` 的目的是防止遮盖 Python 原生的 `list` 函数。这么写可能不美观，但我能想到的其他写法也同样不美观，或者更糟，比如 `my_list`、`the_list`、`list1` 和 `listey` 等。

现在要开始另一个“单元测试 / 编写代码”循环了。

在前几次迭代中，我只给出每次运行测试时期望看到的错误消息，不会告诉你运行测试前要输入哪些代码，你要自己编写每次所需的最少代码改动。

> 需要提示？翻回第 5 章，参照引入 `Item` 模型的步骤。

你首先看到的错误消息是：

```
ImportError: cannot import name 'List'
```

解决这个错误后，再次运行测试会看到：

```
AttributeError: 'List' object has no attribute 'save'
```

然后会看到：

```
django.db.utils.OperationalError: no such table: lists_list
```

因此需要执行一次 makemigrations 命令：

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0003_list.py
    - Create model List
```

然后会看到：

```
    self.assertEqual(first_saved_item.list, list_)
AttributeError: 'Item' object has no attribute 'list'
```

7.9.1　外键关系

Item 的 list 属性应该怎么实现呢？先天真一点，把它当成 text 属性试试（你可以借机看一下你的实现方式与我的有什么区别）：

lists/models.py

```
from django.db import models

class List(models.Model):
    pass

class Item(models.Model):
    text = models.TextField(default='')
    list = models.TextField(default='')
```

照例，测试会告诉我们需要做一次迁移：

```
$ python manage.py test lists
[...]
django.db.utils.OperationalError: no such column: lists_item.list

$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0004_item_list.py
    - Add field list to item
```

看一下测试结果如何：

```
AssertionError: 'List object' != <List: List object>
```

离成功还有一段距离。请仔细看 != 两边的内容。Django 只保存了 List 对象的字符串形式。若想保存对象之间的关系，要告诉 Django 两个类之间的关系，这种关系使用 ForeignKey 字段表示：

lists/models.py

```
from django.db import models

class List(models.Model):
    pass


class Item(models.Model):
    text = models.TextField(default='')
    list = models.ForeignKey(List, default=None)
```

修改之后也要做一次迁移。既然前一个迁移没用了，就把它删掉吧，换一个新的：

```
$ rm lists/migrations/0004_item_list.py
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0004_item_list.py
    - Add field list to item
```

删除迁移是种危险操作，但偶尔需要这么做，因为不可能从一开始就正确定义模型。如果删除已经用于某个数据库的迁移，Django 就不知道当前状态，因此也就不知道如何运行以后的迁移。只有当你确定某个迁移没被使用时才能将其删除。根据经验，已经提交到 VCS 的迁移决不能删除。

7.9.2 根据新模型定义调整其他代码

再看测试的结果如何：

```
$ python manage.py test lists
[...]
ERROR: test_displays_all_items (lists.tests.ListViewTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_can_save_a_POST_request (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id

Ran 6 tests in 0.021s

FAILED (errors=3)
```

天啊，这么多错误。

可是也有一些好消息。虽然很难看出，不过模型测试通过了。但是三个视图测试出现了重大错误。

出现这些错误是因为我们在待办事项和清单之间建立了关联，在这种关联中，每个待办事项都需要一个父级清单，但是原来的测试和代码并没有考虑到这一点。

不过，这正是测试的目的所在。下面要让测试再次通过。最简单的方法是修改 `ListViewTest`，

为测试中的两个待办事项创建父清单：

lists/tests.py (ch07l031)

```
class ListViewTest(TestCase):

    def test_displays_all_items(self):
        list_ = List.objects.create()
        Item.objects.create(text='itemey 1', list=list_)
        Item.objects.create(text='itemey 2', list=list_)
```

修改之后，失败测试减少到两个，而且都是向 `new_list` 视图发送 POST 请求引起的。使用惯用的技术分析调用跟踪，由错误消息找到导致错误的测试代码，然后再找出相应的应用代码，最终定位到下面这行：

```
File "/.../superlists/lists/views.py", line 9, in new_list
Item.objects.create(text=request.POST['item_text'])
```

这行调用跟踪表明创建待办事项时没有指定父清单。因此，要对视图做类似修改：

lists/views.py

```
from lists.models import Item, List
[...]
def new_list(request):
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect('/lists/the-only-list-in-the-world/')
```

修改之后，测试又能通过了：

```
Ran 6 tests in 0.030s

OK
```

此时你是不是感觉不舒畅？**我们为每个新建的待办事项都指定了所属的清单，但还是集中显示所有待办事项，好像它们都属于同一个清单似的——感觉这么做完全不对**。我知道不对，我也有同样的感觉。我们采用的步进方式与直觉不一致，要求代码从一个可用状态变成另一个可用状态。我总想直接动手一次修正所有问题，而不想把一个奇怪的半成品变成另一个半成品。可是你还记得测试山羊吗？爬山时，你要审慎抉择每一步踏在何处，而且一次只能迈一步，确认脚踩的每一个位置都不会让你跌落悬崖。

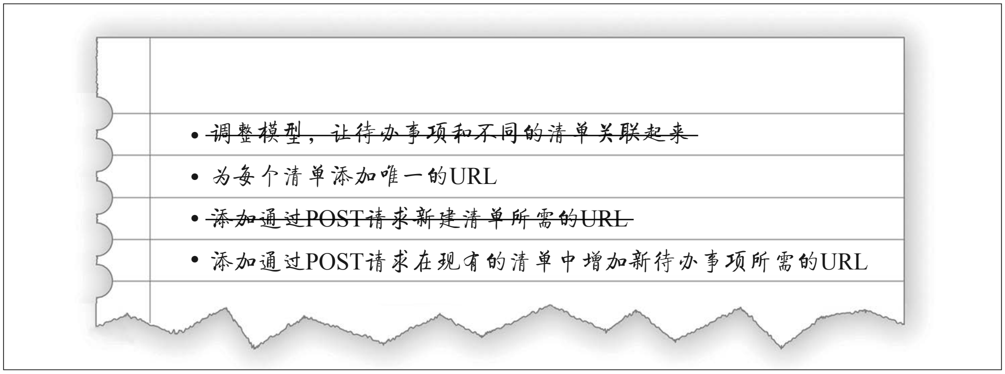
因此，为了确信一切都能正常运行，要再次运行功能测试：

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
[...]
```

毫无疑问，测试的结果和修改前一样。现有功能没有破坏，在此基础上还修改了数据库。这一点令人欣喜！下面提交：

```
$ git status # 改动了3个文件，还新建了2个迁移
$ git add lists
$ git diff --staged
$ git commit
```

又可以从待办事项清单上划掉一个问题了：



7.10　每个列表都应该有自己的URL

应该使用什么作为清单的唯一标识符呢？就目前而言，或许最简单的处理方式是使用数据库自动生成的 `id` 字段。下面修改 `ListViewTest`，让其中的两个测试指向新 URL。

还要把 `test_displays_all_items` 测试重命名为 `test_displays_only_items_for_that_list`，然后在这个测试中确认只显示属于这个清单的待办事项：

lists/tests.py (ch07l033)

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertTemplateUsed(response, 'list.html')

    def test_displays_only_items_for_that_list(self):
        correct_list = List.objects.create()
        Item.objects.create(text='itemey 1', list=correct_list)
        Item.objects.create(text='itemey 2', list=correct_list)
        other_list = List.objects.create()
        Item.objects.create(text='other list item 1', list=other_list)
        Item.objects.create(text='other list item 2', list=other_list)

        response = self.client.get(f'/lists/{correct_list.id}/')

        self.assertContains(response, 'itemey 1')
        self.assertContains(response, 'itemey 2')
        self.assertNotContains(response, 'other list item 1')
        self.assertNotContains(response, 'other list item 2')
```

这个代码清单又用到了几个 f 字符串。如果你还是不太了解，看一下文档（https://docs.python.org/3/reference/lexical_analysis.html#f-strings）。（如果你跟我一样没正式学习过 CS，或许应该跳过正式的语法。）

运行这个单元测试，会看到预期的 404，以及另一个相关的错误：

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)
[...]
FAIL: test_uses_list_template (lists.tests.ListViewTest)
AssertionError: No templates used to render the response
```

7.10.1　捕获URL中的参数

现在要学习如何把 URL 中的参数传入视图：

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/(.+)/$', views.view_list, name='view_list'),
]
```

调整 URL 映射中使用的正则表达式，加入一个**捕获组**（capture group，`.+`），它能匹配随后 / 之前的任意个字符。捕获得到的文本会作为参数传入视图。

也就是说，如果访问 /lists/1/，`view_list` 视图除了常规的 `request` 参数之外，还会获得第二个参数，即字符串 `"1"`。如果访问 /lists/foo/，视图就是 `view_list(request, "foo")`。

但是视图并未期待有参数传入，毫无疑问，这么做会导致问题：

```
ERROR: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
[...]
ERROR: test_uses_list_template (lists.tests.ListViewTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
FAILED (errors=3)
```

这个问题容易修正，在 views.py 中加入一个参数即可：

lists/views.py

```
def view_list(request, list_id):
    [...]
```

现在，前面那个预期失败解决了：

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
AssertionError: 1 != 0 : Response should not contain 'other list item 1'
```

接下来要让视图决定把哪些待办事项传入模板：

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    items = Item.objects.filter(list=list_)
    return render(request, 'list.html', {'items': items})
```

7.10.2　按照新设计调整new_list视图

现在得到发生在另一个测试中的错误：

```
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
ValueError: invalid literal for int() with base 10:
'the-only-list-in-the-world'
```

既然这个测试报错了，就来看看它的代码吧：

lists/tests.py

```
class NewListTest(TestCase):
    [...]

    def test_redirects_after_POST(self):
        response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
        self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
```

看样子这个测试还没按照清单和待办事项的新设计调整，它应该检查视图是否重定向到指定新建清单的 URL：

lists/tests.py (ch07l036-1)

```
    def test_redirects_after_POST(self):
        response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
        new_list = List.objects.first()
        self.assertRedirects(response, f'/lists/{new_list.id}/')
```

修改后测试还是得到**无效字面量**错误。检查一下视图本身，把它改为重定向到有效的地址：

lists/views.py (ch07l036-2)

```
def new_list(request):
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect(f'/lists/{list_.id}/')
```

这样修改之后单元测试就可以通过了：

```
$ python3 manage.py test lists
[...]
......
 ---------------------------------------------------------------------
Ran 6 tests in 0.033s

OK
```

那么功能测试结果如何？差不多也能通过吧？

7.11　功能测试又检测到回归

嗯，快了：

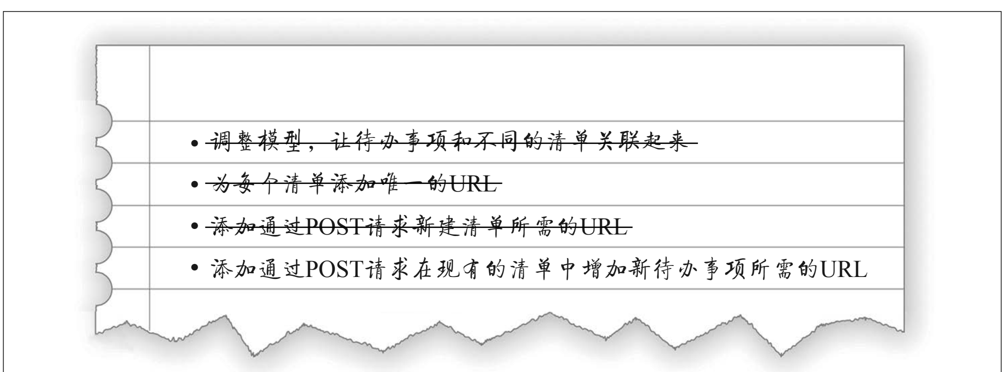
```
F.
======================================================================
FAIL: test_can_start_a_list_for_one_user
(functional_tests.tests.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 67, in
test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']

 ---------------------------------------------------------------------
Ran 2 tests in 8.617s

FAILED (failures=1)
```

新测试其实通过了，不同的用户有不同的清单，但是旧测试提醒我们出现了回归。看起来无法在清单中添加第二个待办事项。这是因为我们投机取巧了，每次 POST 提交都新建一个清单。这正是编写功能测试的目的！

而这正好和待办事项清单中最后一个问题高度吻合：



7.12　还需要一个视图，把待办事项加入现有清单

还需要一个 URL 和视图（/lists/<list_id>/add_item），把新待办事项添加到现有的清单中。我们已经熟知这种操作了，因此可以一次写好两个测试：

lists/tests.py

```
class NewItemTest(TestCase):

    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f'/lists/{correct_list.id}/add_item',
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new item for an existing list')
        self.assertEqual(new_item.list, correct_list)


    def test_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            f'/lists/{correct_list.id}/add_item',
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertRedirects(response, f'/lists/{correct_list.id}/')
```

> 你是不是觉得奇怪，想知道为什么要用 `other_list`？这与查看某个清单的测试类似，要确保把待办事项添加到特定的清单中。在数据库中再存储一个对象便无须使用 `List.objects.first()` 这样可能出错的代码。那样做是不对的，如果真做了，你可能会为之付出惨痛代价（毕竟数字有无穷多个）。这只是主观选择，不过我觉得值得这么做。详情请参见 15.1.1 节。

测试的结果为：

```
AssertionError: 0 != 1
[...]
AssertionError: 301 != 302 : Response didn't redirect as expected: Response
code was 301 (expected 302)
```

7.12.1　小心霸道的正则表达式

有点奇怪，还没在 URL 映射中加入 /lists/1/add_item，应该得到 `404 != 302` 错误，怎么会是 301 呢？

确实令人费解。其实得到这个错误是因为在 URL 映射中使用了一个非常霸道的正则表达式：

superlists/urls.py

```
    url(r'^lists/(.+)/$', views.view_list, name='view_list'),
```

根据 Django 的内部处理机制，如果访问的 URL **几乎**正确，但却少了末尾的斜线，就会得到一个永久重定向响应（301）。在这里，/lists/1/add_item/ 符合 `lists/(.+)/` 的匹配模式，其中 `(.+)` 捕获 `1/add_item`，所以 Django 伸出“援手”，猜测你其实是想访问末尾带斜线的 URL。

这个问题的修正方法是，显式指定 URL 模式只捕获数字，即在正则表达式中使用 `\d`：

superlists/urls.py

```
    url(r'^lists/(\d+)/$', views.view_list, name='view_list'),
```

修改后测试的结果是：

```
AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

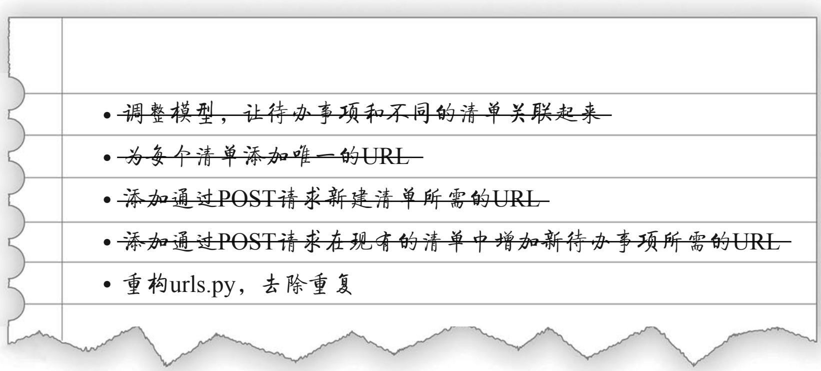
7.12.2　最后一个新URL

现在得到了预期的 404。下面定义一个新 URL，用于把新待办事项添加到现有清单中：

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/(\d+)/$', views.view_list, name='view_list'),
    url(r'^lists/(\d+)/add_item$', views.add_item, name='add_item'),
]
```

现在 URL 映射中定义了三个类似的 URL。在待办事项清单中做个记录，因为这三个 URL 看起来需要重构。



再看测试，现在又提示视图模块缺少属性：

```
AttributeError: module 'lists.views' has no attribute 'add_item'
```

7.12.3　最后一个新视图

定义下面这个视图试试：

lists/views.py

```
def add_item(request):
    pass
```

效果不错：

```
TypeError: add_item() takes 1 positional argument but 2 were given
```

继续修改视图：

lists/views.py

```
def add_item(request, list_id):
    pass
```

测试的结果是：

```
ValueError: The view lists.views.add_item didn't return an HttpResponse object.
It returned None instead.
```

可以从 `new_list` 视图中复制 `redirect`，从 `view_list` 视图中复制 `List.objects.get`：

lists/views.py

```
def add_item(request, list_id):
    list_ = List.objects.get(id=list_id)
    return redirect(f'/lists/{list_.id}/')
```

现在测试的结果为：

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

最后，让视图保存新建的待办事项：

lists/views.py

```
def add_item(request, list_id):
    list_ = List.objects.get(id=list_id)
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect(f'/lists/{list_.id}/')
```

这样，测试又能通过了：

```
Ran 8 tests in 0.050s

OK
```

7.12.4 直接测试响应上下文对象

把待办事项添加到现有清单所需的视图和 URL 都有了，现在只剩在 list.html 模板中使用它们了。打开这个模板，修改表单标签：

lists/templates/list.html

```
<form method="POST" action="but what should we put here?">
```

可是，若想获取添加到当前清单的 URL，模板要知道它渲染的是哪个清单，以及要添加哪些待办事项。希望表单能写成下面这样：

lists/templates/list.html

```
<form method="POST" action="/lists/{{ list.id }}/add_item">
```

为了能这样写，视图要把清单传入模板。下面在 `ListViewTest` 中新建一个单元测试方法：

lists/tests.py (ch07l041)

```
def test_passes_correct_list_to_template(self):
    other_list = List.objects.create()
    correct_list = List.objects.create()
    response = self.client.get(f'/lists/{correct_list.id}/')
    self.assertEqual(response.context['list'], correct_list)   ❶
```

❶ `response.context` 表示要传入 `render` 函数的上下文——Django 测试客户端把上下文附在 `response` 对象上，方便测试。

增加这个测试后得到的结果如下：

```
KeyError: 'list'
```

这是因为没把 `list` 传入模板，其实也给了我们一个简化视图的机会：

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    return render(request, 'list.html', {'list': list_})
```

这么做显然会破坏一个旧测试，因为模板需要 `items`：

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```

可以在 list.html 中修正这个问题，同时还要修改表单 POST 请求的目标地址，即 `action` 属性：

lists/templates/list.html (ch07l043)

```
<form method="POST" action="/lists/{{ list.id }}/add_item">   ❶

  [...]

  {% for item in list.item_set.all %}   ❷
    <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
  {% endfor %}
```

❶ 这是新的目标地址。

❷ `.item_set` 叫作反向查询（reverse lookup），是 Django 提供的非常有用的 ORM 功能，作用是在其他表中查询某个对象的相关记录。

修改模板之后，单元测试能通过了：

```
Ran 9 tests in 0.040s

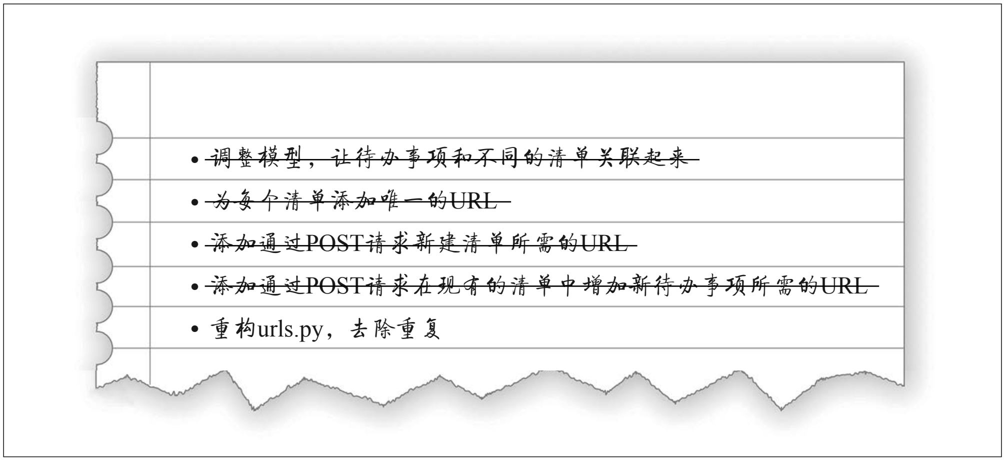
OK
```

功能测试的结果如何呢？

```
$ python manage.py test functional_tests
[...]
..
 ---------------------------------------------------------------------
Ran 2 tests in 9.771s

OK
```

太好了！再看一下待办事项清单：



可惜，测试山羊也是善始不善终的，还有最后一个问题没解决。

在解决这个问题之前，先做提交——着手重构之前一定要提交可正常运行的代码：

```
$ git diff
$ git commit -am "new URL + view for adding to existing lists. FT passes :-)"
```

7.13 使用URL引入做最后一次重构

superlists/urls.py 的真正作用是定义整个网站使用的 URL。如果某些 URL 只在 `lists` 应用中使用，Django 建议使用单独的文件 lists/urls.py，让应用自成一体。定义 `lists` 使用的 URL，最简单的方法是复制现有的 urls.py：

```
$ cp superlists/urls.py lists/
```

然后把 superlists/urls.py 中的三行定义换成一个 `include`：

superlists/urls.py

```
from django.conf.urls import include, url
from lists import views as list_views   ❶
from lists import urls as list_urls   ❶

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)),   ❷
]
```

❶ 顺便使用 `import x as y` 句法为视图和 URL 映射创建别名。在顶层 urls.py 中，这是个好做法，便于从多个应用中引入视图和 URL 映射。其实，后文就会这么做。

❷ 这是那个 `include`。注意，`include` 可以使用一个正则表达式作为 URL 的前缀，这个前缀会添加到引入的所有 URL 前面（这就是去除重复的方法，同时也让代码结构更清晰）。

回到 lists/urls.py 中，我们只需写入那三个 URL 的后半部分，而且不用再写父级 urls.py 中的其他定义：

lists/urls.py(ch07l046)

```
from django.conf.urls import url
from lists import views

urlpatterns = [
    url(r'^new$', views.new_list, name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
    url(r'^(\d+)/add_item$', views.add_item, name='add_item'),
]
```

再次运行单元测试，确认一切仍能正常运行。

我修改时，怀疑自己的能力，不确信能一次改对，所以特意一次只改一个 URL，防止测试失败。如果改错了，还有测试提醒我们。

你可以动手试一下。记得要改回来，而且要确认测试全部都能通过，然后再提交：

```
$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

终于结束了，这一章可真长啊。我们讨论了很多重要话题，先从测试隔离开始，然后又思考了设计。介绍了一些经验法则，比如“YAGNI”和“事不过三，三则重构”。最重要的是，看到了如何一步步修改现有网站，从一个可运行状态变成另一个可运行状态，逐渐实现新设计。

不得不说，我们的网站已经非常接近发布状态了，也就是说，这个待办事项清单网站的首个测试版可以公之于众了。不过，在此之前可能还要做些美化。在接下来的几章中，我们要介绍部署网站时需要做些什么。

更多 TDD 哲学

- 从一个可运行状态到另一个可运行状态（又叫测试山羊与重构猫）
  本能经常驱使我们直接动手一次修正所有问题，但如果不小心，最终可能像重构猫一样，改动了很多代码但都不起作用。测试山羊建议我们一次只迈一步，从一个可运行状态走到另一个可运行状态。

- 把工作分解成易于实现的小任务
  有时，我们要从“乏味的”工作入手，而不是直指有趣的任务。你要相信人只能活一次，平行宇宙中的另一个你可能过得并不好，把功能都破坏了，极尽所能想让应用再次运行起来。

- YAGNI
  “You ain’t gonna need it”（你不需要这个）的简称，劝诫你不要受诱惑编写当时看起来**可能**有用的代码。很有可能你根本用不到这些代码，或者没有准确预见未来的需求。第 22 章给出了一种方法，可以让你避免落入这个陷阱。

第二部分

Web 开发要素

“真正的开发者一定会发布自己的产品。”
——Jeff Atwood

如果这是一本普通编程领域内的 TDD 入门书，到这里我们就可以庆贺一番了，毕竟我们已经掌握了扎实的 TDD 和 Django 基础，也具备了开始开发网站所需的一切知识。

但是，真正的开发者一定会发布自己的产品，那就无法回避 Web 开发中的一些棘手问题，比如静态文件、表单数据验证、可怕的 JavaScript 等，但最令人惧怕的还是部署到生产服务器。

在每个阶段，TDD 都能协助我们正确处理这些问题。

在这一部分中，我仍会尽量让学习曲线保持平缓，而且我们会学到多个重要的新概念和技术。我不会深入展开每个话题，只是希望我所演示的方法足够你在自己的项目中开始使用。如果真想在实际工作中使用这些技术，你还得做些扩展阅读。

如果你开始阅读本书之前并不熟悉 Django，现在花点时间读一遍 Django 官方教程，能很好地巩固目前所学的知识。熟悉 Django 的相关概念后，你会更自信，在接下来的几章中，能专注于我要讲的核心概念。

这一部分有很多有趣的知识，敬请期待！

第 8 章

美化网站：布局、样式及其测试方法

我们正考虑要发布网站的第一个版本，但让人尴尬的是，网站现在看起来还很简陋。本章介绍一些样式基础知识，包括如何集成 Bootstrap 这个 HTML/CSS 框架，还要学习 Django 处理静态文件的方式，以及如何测试静态文件。

8.1　如何在功能测试中测试布局和样式

不可否认，我们的网站现在没有太大的吸引力（如图 8-1）。

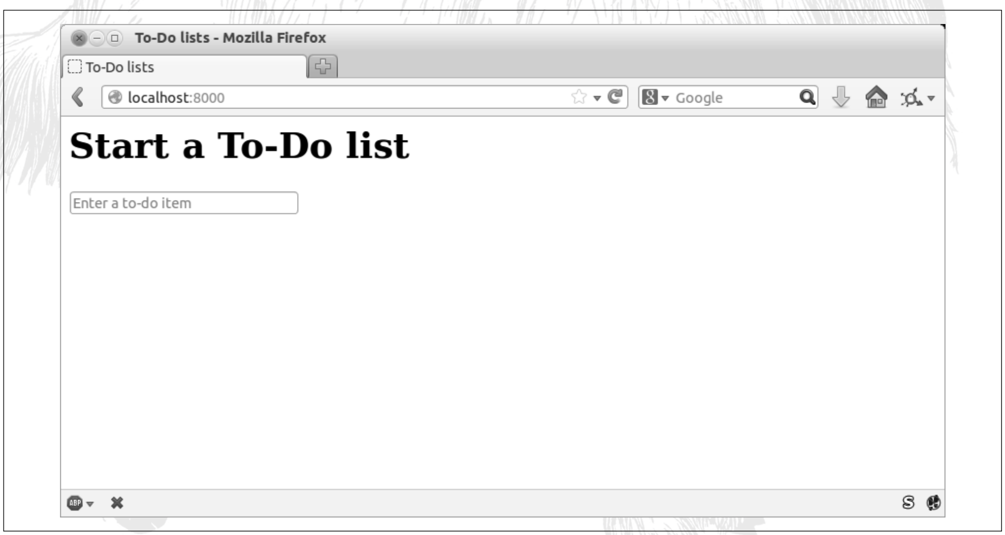


图 8-1：首页，有点简陋

> 执行命令 `manage.py runserver` 启动开发服务器时，可能会看到一个数据库错误：“table lists_item has no column named list_id”（lists_item 表中没有名为 list_id 的列）。此时，需要执行 `manage.py migrate` 命令，更新本地数据库，让 models.py 中的改动生效。如果提醒 `IntegrityErrors`，就删除[1]数据库文件，然后再试。

既然不参加 Python 世界的选丑竞赛，就要美化这个网站。或许我们想实现如下效果。

- 一个精美且很大的输入框，用于新建清单，或者把待办事项添加到现有的清单中。
- 把这个输入框放在一个更大的居中框体中，吸引用户的注意力。

应该怎么使用 TDD 实现这些效果呢？大多数人都会告诉你，不要测试外观。他们是对的，这就像是测试常量一样毫无意义。

但可以测试装饰外观的方式，确信实现了预期的效果即可。例如，使用层叠样式表（Cascading Style Sheet，CSS）编写样式，样式表以静态文件的形式加载，而静态文件配置起来有点儿复杂（稍后会看到，把静态文件移到主机上，配置起来更麻烦），因此只需做某种“冒烟测试”（smoke test），确保加载了 CSS 即可。无须测试字体、颜色以及像素级位置，而是通过简单的测试，确认重要的输入框在每个页面中都按照预期的方式对齐，由此推断页面中的其他样式或许也都正确应用了。

先在功能测试中编写一个新测试方法：

functional_tests/tests.py (ch08l001)

```
class NewVisitorTest(LiveServerTestCase):
    [...]


    def test_layout_and_styling(self):
        # 伊迪丝访问首页
        self.browser.get(self.live_server_url)
        self.browser.set_window_size(1024, 768)

        # 她看到输入框完美地居中显示
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertAlmostEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=10
        )
```

这里有些新知识。先把浏览器的窗口设为固定大小，然后找到输入框元素，获取它的大小和位置，再做些数学计算，检查输入框是否位于网页的中线上。`assertAlmostEqual` 的作用是帮助处理舍入误差以及偶尔由滚动条等事物导致的异常，这里指定计算结果在正负 10 像素范围内为可接受。

注 1：什么？删除数据库？疯了吗？并没有。在开发的过程中，本地开发数据库经常与迁移不同步，而且里面也没什么重要数据，因此可以放心删除。不过要谨慎对待生产服务器中的数据库，详情请参见附录 D。

运行功能测试会得到如下结果：

```
$ python manage.py test functional_tests
[...]
.F.
======================================================================
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 129, in
test_layout_and_styling
    delta=10
AssertionError: 107.0 != 512 within 10 delta

 ---------------------------------------------------------------------
Ran 3 tests in 9.188s

FAILED (failures=1)
```

这次失败在预料之中。不过，这种功能测试很容易出错，所以要用一种有点作弊的快捷方法确认输入框居中时功能测试能通过。一旦确认功能测试编写正确之后，就把这些代码删掉：

lists/templates/home.html (ch08l002)

```
<form method="POST" action="/lists/new">
  <p style="text-align: center;">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
  </p>
  {% csrf_token %}
</form>
```

修改之后测试能通过，说明功能测试起作用了。下面扩展这个测试，确保新建清单后输入框仍然居中对齐显示：

functional_tests/tests.py (ch08l003)

```
        # 她新建了一个清单，看到输入框仍完美地居中显示
        inputbox.send_keys('testing')
        inputbox.send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table('1: testing')
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertAlmostEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=10
        )
```

这会导致测试再次失败：

```
  File "/.../superlists/functional_tests/tests.py", line 141, in
test_layout_and_styling
    delta=10
AssertionError: 107.0 != 512 within 10 delta
```

现在只提交功能测试：

```
$ git add functional_tests/tests.py
$ git commit -m "first steps of FT for layout + styling"
```

现在，似乎找到了满足需求的适当解决方案，能更好地样式化网站。那么就退回添加 `<p style="text-align: center">` 之前的状态吧：

```
$ git reset --hard
```

> `git reset --hard` 是一个破坏力极强的 Git 命令，它会还原所有没提交的改动，所以使用时要小心。它和几乎所有其他 Git 命令都不同，执行之后无法撤销操作。

8.2 使用CSS框架美化网站

设计不简单，现在更难，因为要处理手机、平板等设备。所以很多程序员，尤其是像我一样的懒人，都转而使用 CSS 框架解决问题。框架有很多，不过出现最早且最受欢迎的是 Twitter 开发的 Bootstrap。我们就使用这个框架。

Bootstrap 可在 http://getbootstrap.com/ 获取。

下载 Bootstrap，把它放在 `lists` 应用中一个新文件夹 static 里：[2]

```
$ wget -O bootstrap.zip https://github.com/twbs/bootstrap/releases/download/\
v3.3.4/bootstrap-3.3.4-dist.zip
$ unzip bootstrap.zip
$ mkdir lists/static
$ mv bootstrap-3.3.4-dist lists/static/bootstrap
$ rm bootstrap.zip
```

dist 文件夹中的内容是未经定制的原始 Bootstrap 框架，现在使用这些内容，但在真正的网站中不能这么做，因为用户能立即看出你使用 Bootstrap 时没有定制，业内人士也能由此推知你懒得为网站编写样式。你应该学习如何使用 LESS，至少把字体改了。Bootstrap 文档中有定制的详细说明，或者你可以阅读一篇名为“How to Customize Twitter’s Bootstrap”的指南，写得还不错。

最终得到的 lists 文件夹结构如下：

```
$ tree lists
lists
├── __init__.py
├── __pycache__
│   └── [...]
├── admin.py
├── models.py
├── static
│   └── bootstrap
```

注 2：在 Windows 中不能使用 `wget` 和 `unzip`，但我相信你知道如何下载 Bootstrap，解压缩，再把 dist 文件夹中的内容移到 lists/static/bootstrap 文件夹中。

```
│       ├── css
│       │   ├── bootstrap.css
│       │   ├── bootstrap.css.map
│       │   ├── bootstrap.min.css
│       │   ├── bootstrap-theme.css
│       │   ├── bootstrap-theme.css.map
│       │   └── bootstrap-theme.min.css
│       ├── fonts
│       │   ├── glyphicons-halflings-regular.eot
│       │   ├── glyphicons-halflings-regular.svg
│       │   ├── glyphicons-halflings-regular.ttf
│       │   ├── glyphicons-halflings-regular.woff
│       │   └── glyphicons-halflings-regular.woff2
│       └── js
│           ├── bootstrap.js
│           ├── bootstrap.min.js
│           ├── npm.js
├── templates
│   ├── home.html
│   └── list.html
├── tests.py
├── urls.py
└── views.py
```

在 Bootstrap 文档中的“Getting Started”部分，你会发现 Bootstrap 要求 HTML 模板中包含如下代码：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 101 Template</title>
    <!-- Bootstrap -->
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="http://code.jquery.com/jquery.js"></script>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

我们已经有两个 HTML 模板了，所以不想在每个模板中都添加大量的样板代码。这似乎是运用“不要自我重复”原则的好时机，可以把通用代码放在一起。谢天谢地，Django 使用的模板语言可以轻易做到这一点，这种功能叫作“模板继承”。

8.3　Django模板继承

看一下 home.html 和 list.html 之间的差异：

```
$ diff lists/templates/home.html lists/templates/list.html
<     <h1>Start a new To-Do list</h1>
```

```
<     <form method="POST" action="/lists/new">
---
>     <h1>Your To-Do list</h1>
>     <form method="POST" action="/lists/{{ list.id }}/add_item">
[...]
>     <table id="id_list_table">
>       {% for item in list.item_set.all %}
>         <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
>       {% endfor %}
>     </table>
```

这两个模板头部显示的文本不一样，而且表单的提交地址也不同。除此之外，list.html 还多了一个 `<table>` 元素。

现在我们弄清了两个模板之间共通以及有差异的地方，然后就可以让它们继承同一个父级模板了。先复制 home.html：

```
$ cp lists/templates/home.html lists/templates/base.html
```

把通用的样板代码写入这个基模板中，而且标记出各个“块”，块中的内容留给子模板定制：

lists/templates/base.html

```
<html>
  <head>
    <title>To-Do lists</title>
  </head>

  <body>
    <h1>{% block header_text %}{% endblock %}</h1>
    <form method="POST" action="{% block form_action %}{% endblock %}">
      <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
      {% csrf_token %}
    </form>
    {% block table %}
    {% endblock %}
  </body>
</html>
```

基模板定义了多个叫作“块”的区域，其他模板可以在这些地方插入自己所需的内容。在实际操作中看一下这种机制的用法。修改 home.html，让它继承 base.html：

lists/templates/home.html

```
{% extends 'base.html' %}

{% block header_text %}Start a new To-Do list{% endblock %}

{% block form_action %}/lists/new{% endblock %}
```

可以看出，很多 HTML 样板代码都不见了，只需集中精力编写想定制的部分。然后对 list.html 做同样的修改：

lists/templates/list.html

```
{% extends 'base.html' %}

{% block header_text %}Your To-Do list{% endblock %}

{% block form_action %}/lists/{{ list.id }}/add_item{% endblock %}

{% block table %}
  <table id="id_list_table">
    {% for item in list.item_set.all %}
      <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
    {% endfor %}
  </table>
{% endblock %}
```

对模板来说，这是一次重构。再次运行功能测试，确保没有破坏现有功能：

```
AssertionError: 107.0 != 512 within 10 delta
```

果然，结果和修改前一样。这次改动值得做一次提交：

```
$ git diff -b
# -b的意思是忽略空白，因为我们修改了HTML代码中的一些缩进，所以有必要使用这个旗标
$ git status
$ git add lists/templates # 先不添加static文件夹
$ git commit -m "refactor templates to use a base template"
```

8.4　集成Bootstrap

现在集成 Bootstrap 所需的样板代码更容易了，不过暂时不需要 JavaScript，只加入 CSS 即可：

lists/templates/base.html (ch08l006)

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>To-Do lists</title>
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
[...]
```

行和列

最后，使用 Bootstrap 中某些真正强大的功能。使用之前你得先阅读 Bootstrap 的文档。可以使用栅格系统和 `text-center` 类实现所需的效果：

lists/templates/base.html (ch08l007)

```
<body>
  <div class="container">

    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        <div class="text-center">
          <h1>{% block header_text %}{% endblock %}</h1>
          <form method="POST" action="{% block form_action %}{% endblock %}">
            <input name="item_text" id="id_new_item"
                   placeholder="Enter a to-do item" />
            {% csrf_token %}
          </form>
        </div>
      </div>
    </div>

    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        {% block table %}
        {% endblock %}
      </div>
    </div>

  </div>
</body>
```

（如果你从来没有把一个 HTML 标签分成多行，可能会觉得上述 `<input>` 标签有点儿打破常规。这样写完全可行，如果你不喜欢，可以不这么写。）

> 如果你从未看过 Bootstrap 文档，花点儿时间浏览一下吧。文档中介绍了很多有用的工具，可以运用到你的网站中。

做了上述修改之后，功能测试能通过吗？

```
AssertionError: 107.0 != 512 within 10 delta
```

嗯，还不能。为什么没有加载 CSS 呢？

8.5 Django中的静态文件

Django 处理静态文件时需要知道两件事（其实所有 Web 服务器都是如此）。

(1) 收到指向某个 URL 的请求时，如何区分请求的是静态文件，还是需要经过视图函数处理，生成 HTML。

(2) 到哪里去找用户请求的静态文件。

其实，静态文件就是把 URL 映射到硬盘中的文件上。

为了解决第一个问题，Django 允许我们定义一个 URL 前缀，任何以这个前缀开头的 URL 都被视作针对静态文件的请求。默认的前缀是 /static/，在 settings.py 中定义：

superlists/settings.py

```
[...]

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.11/howto/static-files/

STATIC_URL = '/static/'
```

后面在这一部分添加的设置都和第二个问题有关，即在硬盘中找到真正的静态文件。

既然用的是 Django 开发服务器（`manage.py runserver`），就可以把寻找静态文件的任务交给 Django 完成。Django 会在各应用中每个名为 static 的子文件夹里寻找静态文件。

现在知道把 Bootstrap 框架的所有静态文件都放在 lists/static 文件夹中的原因了吧。可是为什么现在不起作用呢？因为没在 URL 中加入前缀 /static/。再看一下 base.html 中链接 CSS 的元素：

```
<link href="css/bootstrap.min.css" rel="stylesheet">
```

若想让这行代码起作用，要把它改成：

lists/templates/base.html

```
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

现在，开发服务器收到这个请求时就知道请求的是静态文件，因为 URL 以 /static/ 开头。然后，服务器在每个应用中名为 static 的子文件夹里搜寻名为 bootstrap/css/bootstrap.min.css 的文件。最后找到的文件应该是 lists/static/bootstrap/css/bootstrap.min.css。

如果手动在浏览器中查看，应该会看到样式起作用了，如图 8-2 所示。

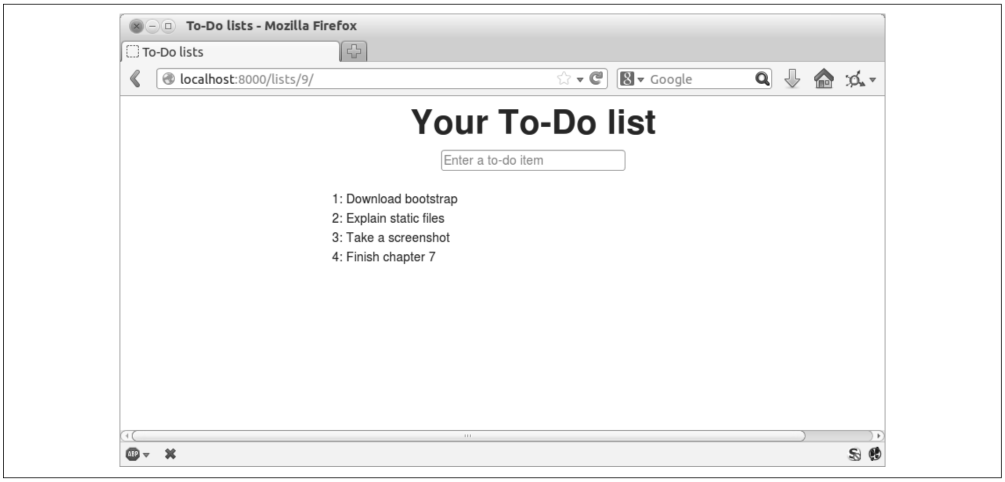


图 8-2：网站开始变得有点好看了

换用StaticLiveServerTestCase

不过，功能测试还无法通过：

```
AssertionError: 107.0 != 512 within 10 delta
```

这是因为，虽然 runserver 启动的开发服务器能自动找到静态文件，但 LiveServerTestCase 找不到。不过无须担心，Django 为开发者提供了一个更神奇的测试类，叫 StaticLiveServerTestCase。

下面换用这个测试类：

functional_tests/tests.py

```
@@ -1,14 +1,14 @@
-from django.test import LiveServerTestCase
+from django.contrib.staticfiles.testing import StaticLiveServerTestCase
 from selenium import webdriver
 from selenium.common.exceptions import WebDriverException
 from selenium.webdriver.common.keys import Keys
 import time

 MAX_WAIT = 10


-class NewVisitorTest(LiveServerTestCase):
+class NewVisitorTest(StaticLiveServerTestCase):

     def setUp(self):
```

现在测试能找到 CSS 了，因此测试也能通过了：

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
...
 ----------------------------------------------------------------------
Ran 3 tests in 9.764s
```

> Windows 用户在这里可能会看到一些错误消息（无关紧要，但容易让人分心）：socket.error: [WinError 10054] An existing connection was forcibly closed by the remote host（现有连接被远程主机强制关闭）。在 tearDown 方法的 self.browser.quit() 之前加上 self.browser.refresh() 就能去掉这些错误。Django 的追踪系统正在关注这个问题。

太好了！

8.6　使用Bootstrap中的组件改进网站外观

看一下使用 Bootstrap 百宝箱中的其他工具能否进一步改善网站的外观。

8.6.1　超大文本块

Bootstrap 中有个类叫 `jumbotron`，用于特别突出地显示页面中的元素。使用这个类放大显示页面的主头部和输入表单：

lists/templates/base.html (ch08l009)

```
<div class="col-md-6 col-md-offset-3 jumbotron">
  <div class="text-center">
    <h1>{% block header_text %}{% endblock %}</h1>
    <form method="POST" action="{% block form_action %}{% endblock %}">
      [...]
```

> 调整网页的设计和布局时，可以打开一个浏览器窗口，时不时地刷新页面。执行 `python manage.py runserver` 命令启动开发服务器，然后在浏览器中访问 http://localhost:8000，边改边看效果。

8.6.2　大型输入框

超大文本块是个好的开始，不过和其他内容相比，输入框显得太小。幸好 Bootstrap 为表单控件提供了一个类，可以把输入框变大：

lists/templates/base.html (ch08l010)

```
<input name="item_text" id="id_new_item"
       class="form-control input-lg"
       placeholder="Enter a to-do item" />
```

8.6.3　样式化表格

现在表格中的文字和页面中的其他文字相比也很小，加上 Bootstrap 提供的 `table` 类可以改进显示效果：

lists/templates/list.html (ch08l011)

```
<table id="id_list_table" class="table">
```

8.7　使用自己编写的CSS

最后，我想让输入表单离标题文字远一点儿。Bootstrap 没有提供现成的解决方案，那么就自己实现吧，引入一个自己编写的 CSS 文件：

lists/templates/base.html

```
[...]
  <title>To-Do lists</title>
  <link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <link href="/static/base.css" rel="stylesheet">
</head>
```

新建文件 lists/static/base.css，写入自己编写的 CSS 新规则。使用输入框的 `id`（`id_new_item`）定位元素，然后为其编写样式：

lists/static/base.css

```
#id_new_item {
    margin-top: 2ex;
}
```

虽然要多操作几步，不过效果很让我满意（如图 8-3）。

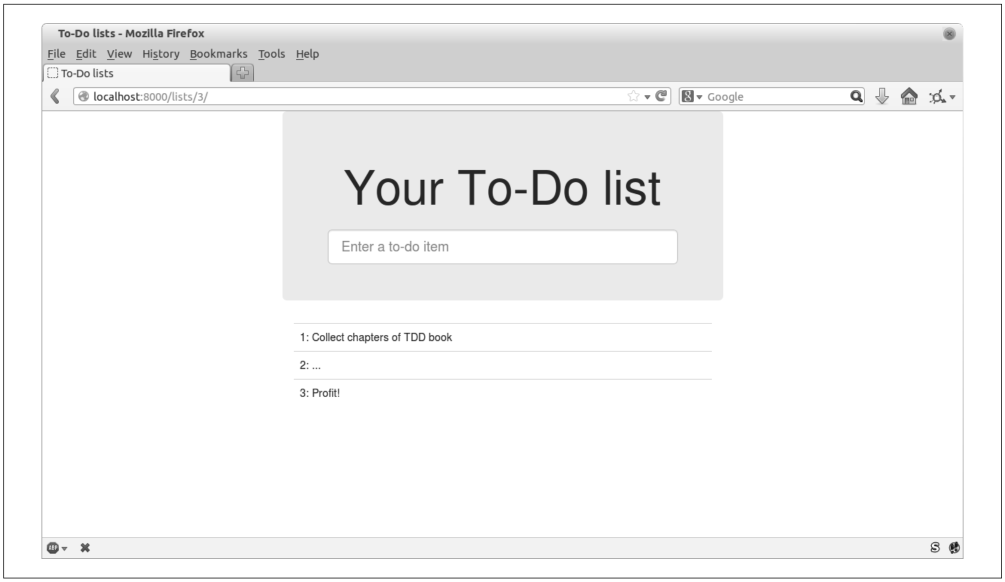


图 8-3：清单页面，各部分都显示得很大

如果想进一步定制 Bootstrap，需要编译 LESS。我**强烈**推荐你花时间定制，总有一天你会有这种需求。LESS、Sass 等其他伪 CSS 类工具，对普通的 CSS 做了很大改进，即便不使用 Bootstrap 也很有用。我不会在这本书中介绍 LESS，网上有很多参考资料，比如说 "How to Customize Twitter's Bootstrap"。

最后再运行一次功能测试，看一切是否仍能正常运行：

```
$ python manage.py test functional_tests
[...]
...
 ---------------------------------------------------------------------
Ran 3 tests in 10.084s

OK
```

样式化暂告段落。现在是提交的绝好时机：

```
$ git status # 修改了tests.py、base.html和list.html，未跟踪lists/static
$ git add .
$ git status # 会显示添加了所有Bootstrap相关文件
$ git commit -m "Use Bootstrap to improve layout"
```

8.8 补遗：collectstatic命令和其他静态目录

前文说过，Django 的开发服务器会自动在应用的文件夹中查找并呈现静态文件。在开发过程中这种功能不错，但在真正的 Web 服务器中，并不需要让 Django 伺服静态内容，因为使用 Python 伺服原始文件速度慢而且效率低，Apache、Nginx 等 Web 服务器能更好地完成这项任务。或许你还会决定把所有静态文件都上传到 CDN（Content Distribution Network，内容分发网络），不放在自己的主机中。

鉴于此，要把分散在各个应用文件夹中的所有静态文件集中起来，复制一份放在一个位置，为部署做好准备。`collectstatic` 命令的作用就是完成这项操作。

静态文件集中放置的位置由 settings.py 中的 `STATIC_ROOT` 定义。下一章会做些部署工作，现在就试着设置一下吧。把 `STATIC_ROOT` 的值设为仓库之外的一个文件夹——我要使用和主源码文件夹同级的一个文件夹：

```
workspace
│    ├── superlists
│    │    ├── lists
│    │    │     ├── models.py
│    │    │
│    │    ├── manage.py
│    │    ├── superlists
│    │
│    ├── static
│    │    ├── base.css
│    │    ├── etc...
```

关键在于，静态文件所在的文件夹不能放在仓库中——不想把这个文件夹纳入版本控制，因为其中的文件和 lists/static 中的一样。

下面是指定这个文件夹位置的一种优雅方式，路径相对 settings.py 文件而言：

superlists/settings.py (ch08l018)

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.11/howto/static-files/

STATIC_URL = '/static/'
STATIC_ROOT = os.path.abspath(os.path.join(BASE_DIR, '../static'))
```

在设置文件的顶部，你会看到定义了 `BASE_DIR` 变量。这个变量利用 `__file__`（这是 Python 内置的变量，特别有用），为我们提供了很大的便利。

下面执行 `collectstatic` 命令试试：

```
$ python manage.py collectstatic
[...]
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap-theme.css'
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap.min.css'

76 static files copied to '/.../static'.
```

如果查看 ../static，会看到所有的 CSS 文件：

```
$ tree ../static/
../static/
├── admin
│   ├── css
│   │   ├── base.css

[...]

│                 └── xregexp.min.js
├── base.css
└── bootstrap
    ├── css
    │   ├── bootstrap.css
    │   ├── bootstrap.css.map
    │   ├── bootstrap.min.css
    │   ├── bootstrap-theme.css
    │   ├── bootstrap-theme.css.map
    │   └── bootstrap-theme.min.css
    ├── fonts
    │   ├── glyphicons-halflings-regular.eot
    │   ├── glyphicons-halflings-regular.svg
    │   ├── glyphicons-halflings-regular.ttf
    │   ├── glyphicons-halflings-regular.woff
    │   └── glyphicons-halflings-regular.woff2
    └── js
        ├── bootstrap.js
        ├── bootstrap.min.js
        └── npm.js

14 directories, 76 files
```

`collectstatic` 命令还收集了管理后台的所有 CSS 文件。管理后台是 Django 的强大功能之一，总有一天我们要知道它的全部功能。但现在还不准备用，所以暂且禁用：

superlists/settings.py

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
]
```

然后再执行 collectstatic 试试：

```
$ rm -rf ../static/
$ python manage.py collectstatic --noinput
Copying '/.../superlists/lists/static/base.css'
[...]
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap-theme.css'
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap.min.css'

15 static files copied to '/.../static'.
```

好多了。

总之，现在知道了怎么把所有静态文件都聚集到一个文件夹中，这样 Web 服务器就能轻易找到静态文件。下一章会深入介绍这个功能和测试方法。

现在，先提交 settings.py 中的改动：

```
$ git diff # 会看到settings.py中的改动
$ git commit -am "set STATIC_ROOT in settings and disable admin"
```

8.9　没谈到的话题

本章只简单介绍了样式化和 CSS，有一些话题我本想涉及，但没做到。你可以进一步研究以下话题。

- 使用 LESS 或 Sass 定制 Bootstrap。
- 使用 {% static %} 模板标签，这样做更符合 DRY 原则，也不用硬编码 URL。
- 客户端打包工具，例如 npm 和 bower。

总结：如何测试设计和布局

简单来说，不应该为设计和布局编写测试。因为这太像是测试常量，所以写出的测试不太牢靠。

这说明设计和布局的**实现过程**极具技巧性，涉及 CSS 和静态文件。因此，可以编写一些简单的“冒烟测试”，确认静态文件和 CSS 起作用即可。下一章我们会看到，把代码部署到生产环境时，冒烟测试能协助我们发现问题。

但是，如果某部分样式需要很多客户端 JavaScript 代码才能使用（我花了很多时间实现动态缩放），就必须为此编写一些测试。

要试着编写最简的测试，确信设计和布局能起作用即可，不必测试具体的实现。我们的目标是能自由修改设计和布局，且无须时不时地调整测试。

第 9 章

使用过渡网站测试部署

“所有乐趣都在部署到生产环境之前。”

——Devops Borat

是时候发布网站的首个版本让公众使用了。人们常说，如果等到一切就绪再发布，等待的时间就太长了。

我们的网站有用吗？是不是比没有要好？能在这个网站上创建待办事项清单吗？这三个疑问的答案都是肯定的。

可是，现在还无法登录，也无法把任务标记为已完成。不过你真的需要这些功能吗？不一定，因为你永远也不知道用户**真正**想使用你的网站做什么。我们觉得用户应该想使用这个网站制定待办事项清单，但他们真正想编写的或许是一个“十佳假蝇钓鱼地”列表，因此就不用提供“标记为已完成”功能。在发布之前，我们永远不知道用户的真实需求。

本章会详细介绍如何把网站部署到真实的线上 Web 服务器中。

你可能想跳过这一章，因为本章有很多令人生畏的知识，或许还觉得部署不是你阅读本书的初衷。但是我**强烈**建议你读一下。本章是我最满意的内容之一，而且有很多读者写信说很庆幸自己当时克服困难读完了这一章。

如果你以前从未把网站部署到服务器上，读过本章后会发现一片新大陆，而且没有什么能比看到自己的网站在真正的互联网中上线更令人满足的了。如果你还迟疑，现今流行的术语，比如“开发运维”（DevOps），一定能让你相信，花时间学习部署是值得的。

> 你的网站上线后请写封信告诉我网址。收到这样的来信，我的心里总是感觉暖暖的。我的电子邮件地址是 obeythetestinggoat@gmail.com。

9.1 TDD以及部署的危险区域

把网站部署到线上 Web 服务器是个很复杂的过程。我们经常会听到这样凄苦的抱怨："但在我的设备中可以正常运行啊！"

部署过程中的一些危险区域如下。

- 静态文件（CSS、JavaScript、图片等）

  Web 服务器往往需要特殊的配置才能伺服静态文件。

- 数据库

  可能会遇到权限和路径问题，还要小心处理，在多次部署之间不能丢失数据。

- 依赖

  要保证服务器上安装了网站依赖的包，而且版本要正确。

不过这些问题都有相应的解决方案。下面一一说明。

- 使用与生产环境一样的基础架构部署**过渡网站**（staging site），这么做可以测试部署的过程，确保部署真正的网站时操作正确。
- 可以在**过渡网站中运行功能测试**，确保服务器中安装了正确的代码和依赖包。而且为了测试网站的布局，我们编写了冒烟测试，这样就能知道是否正确加载了 CSS。
- 与在本地设备上一样，当服务器上运行多个 Python 应用时，可以使用虚拟环境管理包和依赖。
- 最后，一切操作都**自动化**完成。使用自动化脚本部署新版本，使用同一个脚本把网站部署到过渡环境和生产环境，这么做能尽量保证过渡网站和线上网站一样。[1]

在接下来的几页中，我会详细说明一个部署过程。这不是最佳的部署过程，所以别把它当作最佳实践，也别当作是推荐做法。只是做个演示，告诉你部署过程涉及哪些问题，以及测试在这个过程中的作用。

> **内容提要**
>
> 接下来的三章内容很多，这里列出提要，帮助你理清思路。
>
> 本章：搭建基础环境
>
> - 修改功能测试，以便在过渡服务器中运行。
> - 架设服务器，安装所需的全部软件，再把过渡和线上环境使用的域名指向这个服务器。
> - 使用 Git 把代码上传到服务器。
> - 使用 Django 开发服务器在过渡环境的域名下尝试运行过渡网站。
> - 自己动手在服务器上搭建虚拟环境（不用 `virtualenvwrapper`），确保数据库和静态文件都能使用。

注 1：我称之为"过渡"服务器，有人叫它"开发"服务器，还有人叫它"预备生产"服务器。不管叫什么，目的都是架设一个尽量和生产服务器一样的环境来测试代码。

- 在这个过程中不断运行功能测试，找出哪些功能可以正常运行，哪些不能。

下一章：针对生产环境配置

- 修改配置，使其适应生产环境：不再使用 Django 开发服务器，让应用在引导时自动启动，把 DEBUG 设为 False，等等。

关于部署的第三章：自动部署

(1) 配置好之后，编写一个脚本，自动执行前面手动完成的操作，这样以后就能自动部署网站了。

(2) 最后，使用自动化脚本把网站的生产版本部署到真正的域名下。

9.2 一如既往，先写测试

稍微修改一下功能测试，让它能在过渡网站中运行。添加一个参数，指定测试所用的临时服务器地址：

functional_tests/tests.py (ch08l001)

```
import os
[...]

class NewVisitorTest(StaticLiveServerTestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        staging_server = os.environ.get('STAGING_SERVER')  ❶
        if staging_server:
            self.live_server_url = 'http://' + staging_server  ❷
```

还记得我说过 LiveServerTestCase 有一定的缺陷吗？其中一个缺陷是，它总是假定你想使用它自己的测试服务器，这个服务器的地址通过 self.live_server_url 获取。有时我确实想使用这个测试服务器，但也想有别的选择，比如使用一个真正的服务器。

❶ 我通过环境变量 STAGING_SERVER 决定使用哪个服务器。

❷ 我采用的措施是，把 self.live_server_url 替换成"真实"服务器的地址。

按照"常规的"方式运行功能测试，确保上述改动没有破坏现有功能：

```
$ python manage.py test functional_tests
[...]
Ran 3 tests in 8.544s

OK
```

然后指定过渡服务器的 URL 再运行试试。我要使用的过渡服务器地址是 superlists-staging.ottg.eu：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests

======================================================================
FAIL: test_can_start_a_list_for_one_user
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 49, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'Domain name registration | Domain names
| Web Hosting | 123-reg'
[...]

======================================================================
FAIL: test_multiple_users_can_start_lists_at_different_urls
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File
"/.../superlists/functional_tests/tests.py", line 86, in
test_layout_and_styling
    inputbox = self.browser.find_element_by_id('id_new_item')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]
[...]

======================================================================
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]
[...]

Ran 3 tests in 19.480s:

FAILED (failures=3)
```

如果在 Windows 上看到“STAGING_SERVER is not recognized as a command”这样的错误，可能是因为你没使用 Git-Bash。请再看一下“准备工作和应具备的知识”。

可以看到，和预期一样，两个测试都失败了，因为我还没有域名呢。实际上，从第一个调用跟踪中可以看出，访问域名注册商的网站首页时测试就失败了。

不过，看起来功能测试的测试对象是正确的，所以做一次提交吧：

```
$ git diff # 会显示functional_tests.py中的改动
$ git commit -am "Hack FT runner to be able to test staging"
```

别使用 export 设定 STAGING_SERVER 环境变量，否则在当前终端里运行的后续测试都会在过渡网站中运行（如果这跟你的预期不一样，会让你十分困惑）。最好在每次运行功能测试时明确设定。

9.3 注册域名

读到这里，我们需要注册几个域名，不过也可以使用同一个域名下的多个二级域名。我要使用的域名是 superlists.ottg.eu 和 superlists-staging.ottg.eu。如果你还没有域名，现在就得注册一个。再次说明，我真的希望你**按我所说的做**。如果你从未注册过域名，随便选一个老牌注册商买一个便宜的就行，只要花 5 美元左右，甚至还能找到免费域名。我说过想在真正的网站中看到你的应用，这或许能推动你去注册一个域名。

9.4 手动配置托管网站的服务器

可以把部署的过程分成两个任务。

- **配置**新服务器，用于托管代码。
- 把新版代码**部署**到配置好的服务器中。

有些人喜欢每次部署都用全新服务器，我们在 PythonAnywhere 就是这么做的。不过这种做法只适用于大型的复杂网站，或者对现有网站做了重大修改。对我们这个简单的网站来说，分别完成上述两个任务更合理。虽然最终这两个任务都要完全自动化，但就目前而言或许更适合手动配置。

在阅读过程中，你要记住，配置的过程各异，因此部署有很多通用的最佳实践。所以，与其尝试记住我的具体做法，不如试着理解其中的基本原理，这样以后你遇到具体问题时就能使用相同的思想解决了。

9.4.1 选择在哪里托管网站

现今，托管网站有大量不同的方案，不过基本上可以归纳为两类。

- 运行自己的服务器（可能是虚拟服务器）。
- 使用“平台即服务”（Platform-As-A-Service，PaaS）提供商，例如 Heroku、OpenShift 或 PythonAnywhere。

对小型网站而言，PaaS 的优势尤其明显，我强烈建议你考虑使用 PaaS。不过，基于几个原因，本书不会使用 PaaS。首先，有利益冲突，我觉得 PythonAnywhere 是最棒的，但我这么说是因为我在这家公司工作。其实，各家 Paas 提供商提供的支持各不相同，部署的过程也不一样，所以学会其中一家的部署方法并不能在别家使用。任何一家提供商都有可能完全修改部署过程，或者当你阅读这本书时已经停业了。

因此，我们要学习一些优秀的老式服务器管理方法，包括 SSH 和 Web 服务器配置。这些方法永远不会过时，而且学会这些方法还能从头发花白的老前辈那儿得到一些尊重。

我要试着搭建一个十分类似于 PaaS 环境的服务器，不管以后你选择哪种配置方案，都能用到这个部署过程中学到的知识。

9.4.2　搭建服务器

我不规定你该怎么搭建服务器，不管你选择使用 Amazon AWS、Rackspace 或 Digital Ocean，还是自己的数据中心里的服务器，抑或楼梯后橱柜里的 Raspberry Pi——哪种方案都行，只要满足以下条件即可。

- 服务器的系统使用 Ubuntu16.04（“Xenial/LTS”）。
- 有访问服务器的 root 权限。
- 外网可访问。
- 可以通过 SSH 登录。

我推荐使用 Ubuntu 发行版是因为其中安装了 Python 3.6，而且有一些配置 Nginx 的特殊方式（后文会用到）。如果你知道自己在做什么，或许可以换用其他发行版，但遇到问题只能靠自己了。

如果你从未用过 Linux 服务器，完全不知道从何处入手，可以参照我在 GitHub 上写的一篇简要指南（https://github.com/hjwp/Book-TDD-Web-Dev-Python/blob/master/server-quickstart.md）。

> 有些人阅读本章时会跳过购买域名和架设真正的服务器这两部分，直接在自己的电脑中使用虚拟机。请不要这么做，这两种方式**不一样**。配置服务器本身已经很复杂了，如果用虚拟机，阅读本章的内容时会遇到更大的困难。如果你担心要花钱，可以四处找找，域名和服务器都有免费的。如果需要进一步指导，可以给我发电子邮件，我一直都乐于助人。

9.4.3　用户账户、SSH和权限

本节假定你的用户账户没有 root 权限，但有使用 sudo 的权限，因此执行需要 root 权限的操作时，我们可以使用 sudo。在下面的说明中，如果需要用 sudo，我会指出来。

我使用的用户名是“elspeth”，你可以根据自己的喜好起名。

9.4.4　安装Nginx

我们需要一个 Web 服务器，既然现在酷小孩都使用 Nginx，那我们也用它吧。我和 Apache 斗争了多年，可以说单就配置文件的可读性这一项而言，Nginx 如同神赐般拯救了我们。

在我的服务器中安装 Nginx 只需执行一次 `apt-get` 命令即可，然后再执行一个命令就能看到 Nginx 默认的欢迎页面：

```
elspeth@server:$ sudo apt-get install nginx
elspeth@server:$ sudo systemctl start nginx
```

（你可能要先执行 `apt-get update` 和 / 或 `apt-get upgrade`。）

注意本章命令行代码清单中的 elspeth@server，它表示命令必须在服务器中执行，而不是在你自己的电脑中执行。

现在访问服务器的 IP 地址就能看到 Nginx 的“Welcome to nginx”（欢迎使用 Nginx）页面，如图 9-1 所示。



图 9-1：Nginx 可用了

如果没看到这个页面，可能是因为防火墙没有开放 80 端口。以 AWS 为例，你可能得配置服务器的“Security Group”才能打开 80 端口。

9.4.5　安装Python 3.6

写作本书时，Ubuntu 的标准仓库中还没有 Python 3.6，但是用户贡献的“Deadsnakes PPA”中有。

既然我们有 root 权限，下面就来安装所需的系统级关键软件：Python、Git、pip 和 virtualenv。

```
elspeth@server:$ sudo add-apt-repository ppa:fkrull/deadsnakes
elspeth@server:$ sudo apt-get update
elspeth@server:$ sudo apt-get install python3.6 python3.6-venv
```

顺便把 Git 也安装上：

```
elspeth@server:$ sudo apt-get install git
```

9.4.6　解析过渡环境和线上环境所用的域名

不想总是使用 IP 地址，所以要把过渡环境和线上环境所用的域名解析到服务器上。我的注册商提供的控制面板如图 9-2 所示。

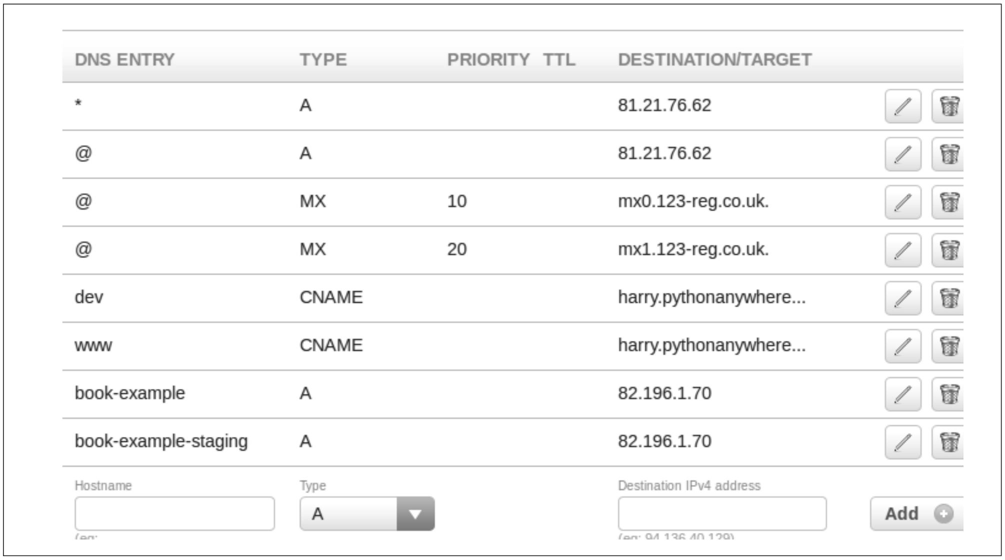


图 9-2：解析域名

在 DNS 系统中，把域名指向一个确切的 IP 地址叫作“A 记录”。各注册商提供的界面有所不同，但在你的注册商网站中四处点击几次应该就能找到正确的页面。

9.4.7　使用功能测试确认域名可用而且Nginx正在运行

为了确认一切顺利，可以再次运行功能测试。你会发现失败消息稍微有点儿不同，其中一个消息和 Nginx 有关：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]
[...]
AssertionError: 'To-Do' not found in 'Welcome to nginx!'
```

有进展。鼓励一下自己，你可以泡杯茶，吃块巧克力饼干。

9.5　手动部署代码

接着要让过渡网站运行起来，检查 Nginx 和 Django 之间能否通信。从这一步起，配置结束了，进入“部署”阶段。在部署的过程中，要思考如何自动化这些操作。

区分配置阶段和部署阶段有个经验法则：配置时需要 root 权限，但部署时不需要。

需要一个文件夹来存放源码。我们把这个目录放在一个非 root 用户的家目录中，在我的服务器中，路径是 /home/elspeth（好像所有共享主机的系统都是这么设置的。不论使用什么主机，一定要以非 root 用户身份运行 Web 应用）。我要按照下面的文件结构存放网站的代码：

```
/home/elspeth
├── sites
│   ├── www.live.my-website.com
│   │    ├── database
│   │    │     └── db.sqlite3
│   │    ├── source
│   │    │    ├── manage.py
│   │    │    ├── superlists
│   │    │    ├── etc...
│   │    │
│   │    ├── static
│   │    │    ├── base.css
│   │    │    ├── etc...
│   │    │
│   │    └── virtualenv
│   │         ├── lib
│   │         ├── etc...
│   │
│   ├── www.staging.my-website.com
│   │    ├── database
│   │    ├── etc...
```

每个网站（过渡网站，线上网站或其他网站）都放在各自的文件夹中。在各文件夹中又有单独的子文件夹，分别存放源码、数据库和静态文件。采用这种结构的逻辑依据是，不同版本的网站源码可能会变，但数据库始终不变。静态文件夹也在同一个相对位置，即 ../static，前一章末尾我们已经设置好了。最后，virtualenv 也有自己的子文件夹（在服务器上无须使用 `virtualenvwrapper`，我们将动手创建虚拟环境）。

9.5.1 调整数据库的位置

首先，在 settings.py 中修改数据库的位置，而且要保证修改后的位置在本地电脑中也能使用：

superlists/settings.py (ch08l003)

```
# 项目内的路径使用os.path.join(BASE_DIR, ...)形式构建
import os
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
[...]
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, '../database/db.sqlite3'),
    }
}
```

看一下 settings.py 文件顶部 `BASE_DIR` 的定义。注意，最内层是 `abspath`。处理路径时一定要这么做，否则导入文件时会遇到各种奇怪的问题。感谢 Green Nathan 指出这一点。

在本地试一下：

```
$ mkdir ../database
$ python manage.py migrate --noinput
Operations to perform:
Apply all migrations: auth, contenttypes, lists, sessions
Running migrations:
[...]
$ ls ../database/
db.sqlite3
```

看起来可以正常使用。做次提交：

```
$ git diff # 会看到settings.py中的改动
$ git commit -am "move sqlite database outside of main source tree"
```

借助代码分享网站，使用 Git 把代码上传到服务器。如果之前没推送，现在要把代码推送到 GitHub、BitBucket 或同类网站中。这些网站都为初学者提供了很好的用法说明，告诉你该怎么推送。

把源码上传到服务器所需的全部 Bash 命令如下所示。如果你不熟悉 Bash 命令，我告诉你，`export` 的作用是创建一个可在 bash 中使用的“本地变量”：

```
elspeth@server:$ export SITENAME=superlists-staging.ottg.eu
elspeth@server:$ mkdir -p ~/sites/$SITENAME/database
elspeth@server:$ mkdir -p ~/sites/$SITENAME/static
elspeth@server:$ mkdir -p ~/sites/$SITENAME/virtualenv
# 要把下面这行命令中的URL换成你自己仓库的URL
elspeth@server:$ git clone https://github.com/hjwp/book-example.git \
~/sites/$SITENAME/source
Resolving deltas: 100% [...]
```

使用 `export` 定义的 Bash 变量只在当前终端会话中有效。如果退出服务器后再登录，就需要重新定义。这个特性有点隐晦，因为 Bash 不会报错，而是直接用空字符串表示未定义的变量，这种处理方式会导致诡异的结果。如果不信，可以执行 **`echo $SITENAME`** 试试。

现在网站安装好了，在开发服务器中运行试试——这是一个冒烟测试，检查所有活动部件是否连接起来了：

```
elspeth@server:$ $ cd ~/sites/$SITENAME/source
$ python manage.py runserver
Traceback (most recent call last):
  File "manage.py", line 8, in <module>
    from django.core.management import execute_from_command_line
ImportError: No module named django.core.management
```

啊，服务器上还没安装 Django。

9.5.2　手动创建虚拟环境，使用requirements.txt

为了“保存”虚拟环境所需的包列表，也为了能够重建服务器，我们要创建 requirements.txt 文件：

```
$ echo "django==1.11" > requirements.txt
$ git add requirements.txt
$ git commit -m "Add requirements.txt for virtualenv"
```

> 你可能觉得奇怪，为什么没在需要的依赖列表中添加 Selenium？这是因为 Selenium 只是测试的依赖，而不是应用代码的依赖。有些人喜欢再创建一个名为 test-requirements.txt 的文件。

现在执行 `git push` 命令，把更新推送到代码分享网站：

```
$ git push
```

然后，把改动拉取到服务器上：

```
elspeth@server:$ git pull  # 可能会让你先对git做些配置
```

若想手动创建虚拟环境（即不使用 `virtualenvwrapper`），就要使用标准库中的 `venv` 模块，指定虚拟环境的存放路径：

```
elspeth@server:$ pwd
/home/espeth/sites/staging.superlists.com/source
elspeth@server:$ python3.6 -m venv ../virtualenv
elspeth@server:$ ls ../virtualenv/bin
activate      activate.fish  easy_install-3.6  pip3    python
activate.csh  easy_install   pip               pip3.6  python3
```

如果想激活这个虚拟环境，就执行 `source ../virtualenv/bin/activate`，但是无须这么做。其实，可以直接调用虚拟环境 bin 目录中的可执行文件，运行相应版本的 Python、pip 等。稍后就将这么做。

为了把所需的依赖安装到虚拟环境中，使用虚拟环境中的 pip：

```
elspeth@server:$ ../virtualenv/bin/pip install -r requirements.txt
Downloading/unpacking Django==1.11 (from -r requirements.txt (line 1))
[...]
Successfully installed Django
```

为了运行虚拟环境中的 Python，使用虚拟环境中的 python 二进制文件：

```
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
Validating models...
0 errors found
[...]
```

> 如果防火墙配置得当，你现在甚至可以手动访问网站。你要执行 `runserver 0.0.0.0:8000`，监听外网和内网 IP 地址，然后访问 http://your.domain.com:8000。

看起来能正常运行。按 Ctrl-C 键停止服务器。

又取得了进展！现在，我们能把代码推送到服务器上（`git push`），也能从服务器上拉取代码（`git pull`）。而且我们搭建了一个与本地一致的虚拟环境，还有一个文件（requirements.txt）同步依赖。

接下来将配置 Nginx Web 服务器，让它与 Django 通信，并把网站放到标准的 80 端口上。

9.5.3　简单配置Nginx

下面创建一个 Nginx 配置文件，把过渡网站收到的请求交给 Django 处理。如下是一个极简的配置。

server: /etc/nginx/sites-available/superlists-staging.ottg.eu

```
server {
    listen 80;
    server_name superlists-staging.ottg.eu;

    location / {
        proxy_pass http://localhost:8000;
    }
}
```

这个配置只监听过渡网站的域名，而且会把所有请求“代理”到本地 8000 端口，等待 Django 处理请求后得到的响应。

我把这个配置保存为 superlists-staging.ottg.eu 文件，放在 /etc/nginx/sites-available 文件夹里。

> 不知道在服务器中如何编辑文件吗？服务器中都有 vi，我建议你之后学习一下如何使用这个工具。此外，也可以使用对初学者相对友好的 `nano`。注意，还得使用 `sudo`，因为这个文件在系统文件夹中。

然后创建一个符号链接，把这个文件加入启用的网站列表中：

```
elspeth@server:$ echo $SITENAME # 检查在这个shell会话中是否还能使用这个变量获取网站名
superlists-staging.ottg.eu
elspeth@server:$ sudo ln -s ../sites-available/$SITENAME /etc/nginx/sites-
enabled/$SITENAME
elspeth@server:$ ls -l /etc/nginx/sites-enabled # 确认符号链接是否在那里
```

在 Debian 和 Ubuntu 中，这是保存 Nginx 配置的推荐做法——把真正的配置文件放在 sites-available 文件夹中，然后在 sites-enabled 文件夹中创建一个符号链接。这么做便于切换网站的在线状态。

或许我们还可以把默认的“Welcome to nginx”页面删除，避免混淆：

```
elspeth@server:$ sudo rm /etc/nginx/sites-enabled/default
```

现在测试一下配置：

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
```

我还要编辑 /etc/nginx/nginx.conf 文件，把 `server_names_hash_bucket_size 64;` 这行的注释去掉，这样才能使用我的长域名。你或许不会遇到这个问题。执行 `reload` 命令时，如果配置文件有问题，Nginx 会提醒你。

快速进行视觉确认——网站运行起来了（如图 9-3）！

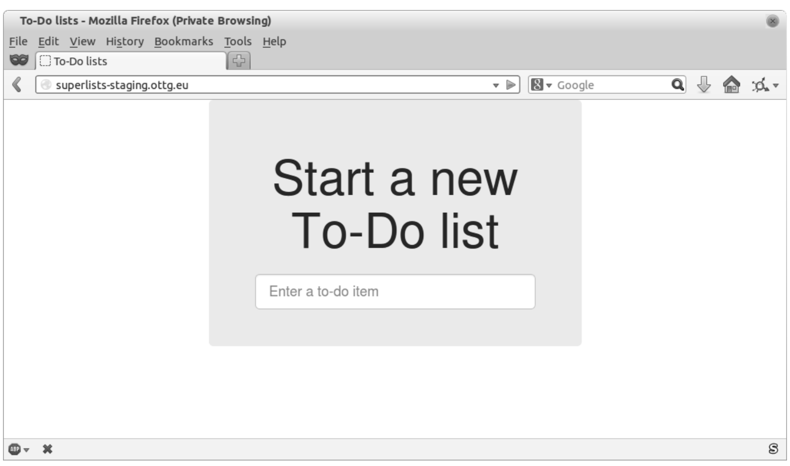


图 9-3：过渡网站运行起来了！

如果 Nginx 出现异常，执行 `sudo nginx -t` 命令试试。这个命令的作用是测试配置；如果发现配置文件中有问题，它会提醒你。

我们来看功能测试的结果如何：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
[...]
AssertionError: 0.0 != 512 within 3 delta
```

尝试提交新待办事项时测试失败了，因为还没设置数据库。运行测试时你可能注意到了 Django 的黄色报错页（如图 9-4 所示，手动访问网站也能看到），页面中显示的信息和测试失败消息差不多。

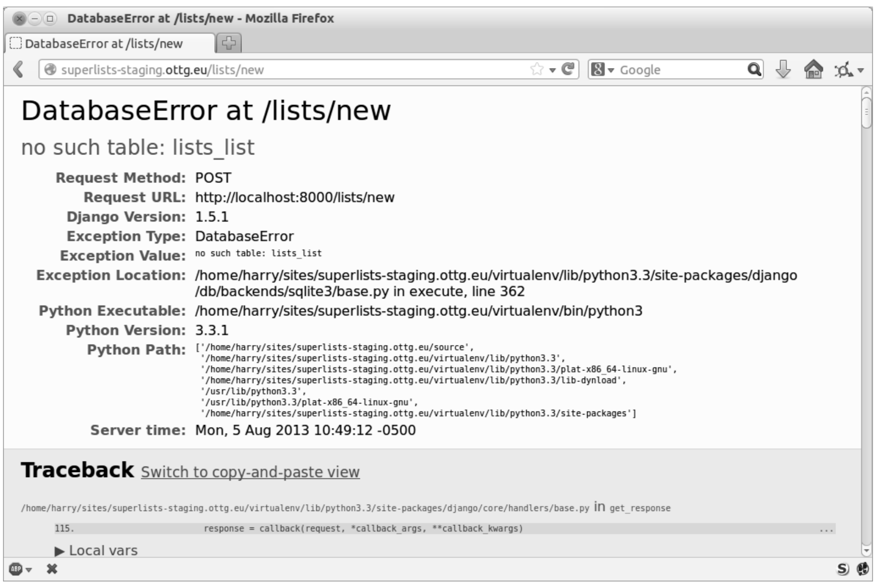


图 9-4：数据库还无法使用

> 测试避免让我们陷入可能出现的窘境之中。访问网站首页时**看起来**很正常，如果此时草率地认为工作结束了，那个扰人的 Django 报错页就会被网站的首批用户发现。好吧，这么说可能夸大了影响，说不定我们自己**已经**发现了，可是如果网站越来越大、越来越复杂怎么办？你不可能确认每项功能，但测试能。

9.5.4　使用迁移创建数据库

执行 `migrate` 命令时，可以指定 `--noinput` 参数，禁止两次询问“你确定吗”：

```
elspeth@server:$ ../virtualenv/bin/python manage.py migrate --noinput
Creating tables ...
[...]
```

```
elspeth@server:$ ls ../database/
db.sqlite3
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
```

再运行功能测试试试：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]

...
 ---------------------------------------------------------------------
Ran 3 tests in 10.718s

OK
```

看到网站运行起来的感觉太棒了！继续阅读下一节之前，或许我们可以犒赏自己，喝杯茶休息一下——这是你应得的奖励。

> 如果看到“502 - Bad Gateway”错误，可能是因为执行 `migrate` 命令之后忘记使用 `manage.py runserver` 重启开发服务器。

下述框注中还有一些调试技巧。

> **服务器调试技巧**
>
> 部署是个棘手活儿。如果遇到问题，可以使用以下技巧找出原因。
>
> - 我知道你已经检查过了，不过还是再检查一遍各个文件，看它们的位置和内容是否正确。哪怕只错一个字符，也可能导致重大问题。
> - 查看 Nginx 的错误日志，存储在 /var/log/nginx/error.log 中。
> - 可以使用 `-t` 标志检查 Nginx 的配置：`nginx -t`。
> - 确保浏览器没有缓存过期的响应。按下 Ctrl 键的同时点击刷新按钮，或者打开一个新的隐私窗口。
> - 最后，可以使用 `sudo reboot` 彻底重启试试。遇到无从下手的问题时，我有时就是这样解决的。
>
> 如果真的遇到无法解决的问题，还可以推倒重来。第二次肯定顺手得多……

9.6 手动部署大功告成

终于结束了。经过一番努力，我们让网站运行起来了，至少基本上可用了。但是，在生产环境真的不能使用 Django 开发服务器，而且不能靠手动执行 `runserver` 命令启动服务器。下一章将改进部署过程，让它更适用于生产环境。

测试驱动服务器配置和部署

- 测试能降低部署过程的不确定性
  对开发者来说，管理服务器始终充满“乐趣”，我的意思是，这个过程充满不确定性和意外。我写这一章的目的是告诉你，功能测试组件能降低这个过程的不确定性。
- 常见的痛点——数据库、静态文件、依赖、自定义设置
  数据库配置、静态文件、软件依赖和自定义设置在开发环境和生产环境之间有所不同，部署时要格外留意。自己部署时，一定要审慎处理这些事物。
- 有测试护航，可以放心试验
  改动服务器配置后，可以运行测试组件，确保一切依然正常。有测试的保护，我们可以放心试验，少一分担忧（具体内容参见下一章）。

第10章

为部署到生产环境做好准备

本章将做些修改，让我们的网站更适应生产环境的配置。每次修改都将通过测试确认功能是否依然可用。

前面的部署过程有什么问题呢？问题在于，生产环境不能使用 Django 开发服务器，而且没有考虑到“真实的”负载。我们将换用 Gunicorn 运行 Django 代码，并且使用 Nginx 伺服静态文件。

目前的 settings.py 把 `DEBUG` 设为 `True`，我极不推荐在生产环境这么做（我们可不想在网站出错时让用户看到供调试的调用跟踪）。安全起见，我们还将设定 `ALLOWED_HOSTS`。

我们希望网站在服务器重启后自动启动。为此，我们将编写一个 Systemd 配置文件。

最后，把网站绑定到 8000 端口会导致无法在服务器中运行多个网站，因此将换用“unix 套接字”连通 Nginx 和 Django。

10.1　换用Gunicorn

知道为什么 Django 的吉祥物是一匹小马吗？ Django 提供了很多功能，包括 ORM、各种中间件、网站后台等。“除了小马之外你还想要什么呢？”我想，既然你已经有一匹小马了，或许你还想要一头“绿色独角兽”（Green Unicorn），即 Gunicorn。

```
elspeth@server:$ ../virtualenv/bin/pip install gunicorn
```

Gunicorn 需要知道 WSGI（Web Server Gateway Interface，Web 服务器网关接口）服务器的路径。这个路径往往可以使用一个名为 `application` 的函数获取。Django 在文件 superlists/wsgi.py 中提供了这个函数：

```
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
2013-05-27 16:22:01 [10592] [INFO] Starting gunicorn 0.19.6
2013-05-27 16:22:01 [10592] [INFO] Listening at: http://127.0.0.1:8000 (10592)
[...]
```

如果现在访问网站，会发现所有样式都失效了，如图 10-1 所示。

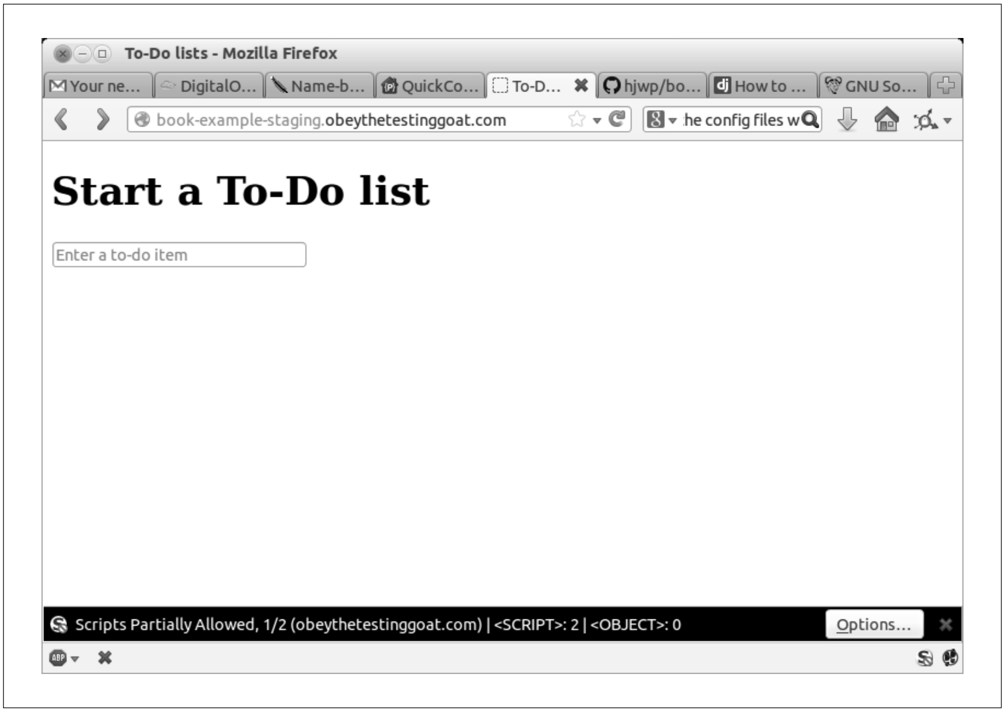


图 10-1：样式失效

如果运行功能测试，会看到的确出问题了。添加待办事项的测试能顺利通过，但布局和样式的测试失败了。测试做得不错！

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]
AssertionError: 125.0 != 512 within 3 delta
FAILED (failures=1)
```

样式失效的原因是，Django 开发服务器会自动伺服静态文件，但 Gunicorn 不会。现在配置 Nginx，让它代为伺服静态文件。

我们前进了一步，又后退了一步，不过有测试在辅助我们。继续前进！

10.2　让Nginx伺服静态文件

首先，执行 `collectstatic` 命令，把所有静态文件复制到一个 Nginx 能找到的文件夹中：

```
elspeth@server:$ ../virtualenv/bin/python manage.py collectstatic --noinput
elspeth@server:$ ls ../static/
base.css bootstrap
```

下面配置 Nginx，让它伺服静态文件：

```
server {
    listen 80;
    server_name superlists-staging.ottg.eu;

    location /static {
        alias /home/elspeth/sites/superlists-staging.ottg.eu/static;
    }

    location / {
        proxy_pass http://localhost:8000;
    }
}
```

然后重启 Nginx 和 Gunicorn：

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
```

如果再次访问网站，会看到外观正常多了。可以再次运行功能测试确认：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]

...
 ---------------------------------------------------------------------
Ran 3 tests in 10.718s

OK
```

搞定！

10.3　换用Unix套接字

如果想要同时伺服过渡网站和线上网站，这两个网站就不能共用 8000 端口。可以为不同网站分配不同端口，但这么做有点儿随意，而且很容易出错，万一在线上网站的端口上启动过渡服务器（或者反过来）怎么办。

更好的方法是使用 Unix 域套接字。域套接字类似于硬盘中的文件，不过还可以用来处理 Nginx 和 Gunicorn 之间的通信。要把套接字保存在文件夹 /tmp 中。下面修改 Nginx 的代理设置。

server: /etc/nginx/sites-available/superlists-staging.ottg.eu

```
[...]
    location / {
        proxy_set_header Host $host;
        proxy_pass http://unix:/tmp/superlists-staging.ottg.eu.socket;
    }
}
```

proxy_set_header 的作用是让 Gunicorn 和 Django 知道它们运行在哪个域名下。ALLOWED_HOSTS 安全功能需要这个设置，稍后会启用这个功能。

现在重启 Gunicorn，不过这一次告诉它监听套接字，而不是默认的端口：

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/gunicorn --bind \
    unix:/tmp/superlists-staging.ottg.eu.socket superlists.wsgi:application
```

还要再次运行功能测试，确保所有测试仍能通过：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]
OK
```

还差几步才能完成部署。

10.4　把DEBUG设为False，设置ALLOWED_HOSTS

在自己的服务器中开启调试模式有利于排查问题，但显示满页的调用跟踪不安全。

在 settings.py 的顶部有 DEBUG 设置项。如果把它设为 False，还需要设置另一个选项，ALLOWED_HOSTS。这个设置在 Django 1.5 中添加，目的是提高安全性。不过，在默认的 settings.py 中没有为这个功能提供有帮助的注释。那就自己添加这个选项吧，在服务器中按照下面的方式修改 settings.py。

server: superlists/settings.py

```
# 安全警告：别在生产环境中开启调试模式!
DEBUG = False

TEMPLATE_DEBUG = DEBUG

# DEBUG=False时需要这项设置
ALLOWED_HOSTS = ['superlists-staging.ottg.eu']
[...]
```

然后重启 Gunicorn，再运行功能测试，确保一切正常。

> 在服务器中别提交这些改动。现在这只是为了让网站正常运行做的小调整，不是需要纳入仓库的改动。一般来说，简单起见，我只会在本地电脑中把改动提交到 Git 仓库中。如果需要把代码同步到服务器中，再使用 git push 和 git pull。

再次运行测试，确认一切正常：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]
OK
```

很好。

10.5 使用Systemd确保引导时启动Gunicorn

部署的最后一步是确保服务器引导时自动启动 Gunicorn，如果 Gunicorn 崩溃了还要自动重启。在 Ubuntu 中，可以使用 Systemd 实现这个功能。

server: /etc/systemd/system/gunicorn-superlists-staging.ottg.eu.service

```
[Unit]
Description=Gunicorn server for superlists-staging.ottg.eu

[Service]
Restart=on-failure  ❶
User=elspeth  ❷
WorkingDirectory=/home/elspeth/sites/superlists-staging.ottg.eu/source  ❸
ExecStart=/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/bin/gunicorn \
    --bind unix:/tmp/superlists-staging.ottg.eu.socket \
    superlists.wsgi:application  ❹

[Install]
WantedBy=multi-user.target  ❺
```

Systemd 的配置很简单（如果曾经编写过 init.d 脚本，会觉得更简单），而且一目了然。

❶ `Restart=on-failure` 指明在崩溃时自动重启进程。

❷ `User=elspeth` 指明以“elspeth”用户的身份运行进程。

❸ `WorkingDirectory` 设定当前工作目录。

❹ `ExecStart` 是要执行的进程。为了提高可读性，我们使用行接续符 \ 把整个命令分成多行，不过也可以写成一行。

❺ `[Install]` 区中的 `WantedBy` 告诉 Systemd，我们想在引导时启动这个服务。

Systemd 脚本保存在 /etc/systemd/system 中，而且文件名必须以 .service 结尾。

下面告诉 Systemd，使用 `systemctl` 命令启动 Gunicorn：

```
# 必须执行这个命令，让Systemd加载新的配置文件
elspeth@server:$ sudo systemctl daemon-reload
# 这个命令让Systemd在引导时加载服务
elspeth@server:$ sudo systemctl enable gunicorn-superlists-staging.ottg.eu
# 这个命令启动服务
elspeth@server:$ sudo systemctl start gunicorn-superlists-staging.ottg.eu
```

（顺便说一下，你会发现 `systemctl` 命令支持按制表符补全，甚至可以补全服务名称。）

现在可以再次运行功能测试，确保一切仍能正常运行。你甚至还可以重启服务器，查看网站能否自动重新运行。

更多调试技巧

- 执行 `sudo journalctl -u gunicorn-superlists-staging.ottg.eu` 命令查看 Systemd 的日志。
- 可以让 Systemd 检查服务配置是否有效：`systemd-analyze verify /path/to/my.service`。
- 改动后记得重启相关服务。
- 修改 Systemd 配置文件后，如果想查看改动的效果，要在执行 `systemctl restart` 之前执行 `daemon-reload` 命令。

保存改动：把Gunicorn添加到requirements.txt

回到**本地**仓库，应该把 Gunicorn 添加到虚拟环境所需的包列表中：

```
$ pip install gunicorn
$ pip freeze | grep gunicorn >> requirements.txt
$ git commit -am "Add gunicorn to virtualenv requirements"
$ git push
```

写作本书时，在 Windows 中使用 `pip` 能顺利安装 Gunicorn，但 Gunicorn 无法正常使用。幸好我们只在服务器中使用 Gunicorn，因此这不是问题。不过，对 Windows 的支持正在讨论中。

10.6　考虑自动化

总结一下配置和部署的过程。

- 配置

  (1) 假设有用户账户和家目录。

  (2) `add-apt-repository ppa:fkrull/deadsnakes`。

  (3) `apt-get install nginx git python3.6 pxthon3.6-venv`。

  (4) 添加 Nginx 虚拟主机配置。

  (5) 添加 Upstart 任务，自动启动 Gunicorn。

- 部署

  (1) 在 ~/sites 中创建目录结构。

  (2) 拉取源码，保存到 source 文件夹中。

  (3) 启用 ../virtualenv 中的虚拟环境。

  (4) `pip install -r requirements.txt`。

  (5) 执行 `manage.py migrate`，创建数据库。

  (6) 执行 `collectstatic` 命令，收集静态文件。

  (7) 在 settings.py 中设置 `DEBUG = False` 和 `ALLOWED_HOSTS`。

(8) 重启 Gunicorn。
(9) 运行功能测试，确保一切正常。

假设现在不用完全自动化配置过程，应该怎么保存现阶段取得的结果呢？我说应该把 Nginx 和 Systemd 配置文件保存起来，便于以后重用。下面把这两个配置文件保存到仓库中一个新建的子文件夹中。

保存配置文件的模板

首先，创建这个子文件夹：

```
$ mkdir deploy_tools
```

下面是 Nginx 配置的通用模板：

deploy_tools/nginx.template.conf

```
server {
    listen 80;
    server_name SITENAME;

    location /static {
        alias /home/elspeth/sites/SITENAME/static;
    }

    location / {
        proxy_set_header Host $host;
        proxy_pass http://unix:/tmp/SITENAME.socket;
    }
}
```

下面是 Gunicorn Systemd 服务的模板：

deploy_tools/gunicorn-systemd.template.service

```
[Unit]
Description=Gunicorn server for SITENAME

[Service]
Restart=on-failure
User=elspeth
WorkingDirectory=/home/elspeth/sites/SITENAME/source
ExecStart=/home/elspeth/sites/SITENAME/virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.socket \
    superlists.wsgi:application

[Install]
WantedBy=multi-user.target
```

再使用这两个文件配置新网站就容易了，查找替换 `SITENAME` 即可。

其他步骤做些笔记就行。为什么不在仓库中建个文件保存说明呢？

deploy_tools/provisioning_notes.md

```
配置新网站
=======================

## 需要的包：

* nginx
* Python 3.6
* virtualenv + pip
* Git

以Ubuntu为例：

    sudo add-apt-repository ppa:fkrull/deadsnakes
    sudo apt-get install nginx git python36 python3.6-venv

## Nginx虚拟主机

* 参考nginx.template.conf
* 把SITENAME替换成所需的域名，例如staging.my-domain.com

## Systemd服务

* 参考gunicorn-upstart.template.conf
* 把SITENAME替换成所需的域名，例如staging.my-domain.com

## 文件夹结构：
假设有用户账户，家目录为/home/username

/home/username
└── sites
    └── SITENAME
         ├── database
         ├── source
         ├── static
         └── virtualenv
```

然后提交上述改动：

```
$ git add deploy_tools
$ git status # 看到3个新文件
$ git commit -m "Notes and template config files for provisioning"
```

现在，源码的目录结构如下所示：

```
.
├── deploy_tools
│   ├── gunicorn-systemd.template.service
│   ├── nginx.template.conf
│   └── provisioning_notes.md
├── functional_tests
│   ├── [...]
├── lists
│   ├── __init__.py
│   ├── models.py
```

```
│   ├── [...]
│   ├── static
│   │   ├── base.css
│   │   └── bootstrap
│   │       ├── [...]
│   ├── templates
│   │   ├── base.html
│   │   ├── [...]
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── manage.py
├── requirements.txt
└── superlists
    ├── [...]
```

10.7 保存进度

在过渡服务器中运行功能测试，能让我们相信网站确实能正常运行。但大多数情况下，你并不想在真正的服务器中运行功能测试。为了不让我们的劳动付之东流，并且保证生产服务器和过渡服务器一样能正常运行，要让部署的过程可重复执行。

为此，我们需要自动化。这是下一章的话题。

为部署生产服务器做好准备

为生产服务器环境做准备时要考虑以下几点。

- 不要在生产环境使用 Django 开发服务器
  Django 更适合在 Gunicorn 或 uWSGI 等中运行，因为它们支持运行多个职程（worker）。
- 不要使用 Django 伺服静态文件
  没必要用 Python 进程处理伺服静态文件这样的简单任务。可以交给 Nginx 去做，当然其他 Web 服务器也能做到，比如 Apache 或 uWSGI。
- 检查 settings.py 中只针对开发环境的设置
  本章提到了 `DEBUG=True` 和 `ALLOWED_HOSTS`，不过可能还有其他设置要注意（让服务器发送电子邮件时还会涉及几个）。
- 安全性
  本书篇幅有限，无法深入讨论服务器安全性。但我要提醒你，若想自己运行服务器，一定要掌握一些安全知识。（有些人选择使用 PaaS 的一个原因就是无须过多担心安全问题。）如果你不知从何着手，可以阅读这篇文章："My first 5 minutes on a server"。强烈建议你安装 fail2ban，然后查看它的日志文件。你会惊奇地发现，尝试暴力登录 SSH 的活动多得不得了。互联网可不是什么安全之所！

第 11 章

使用Fabric自动部署

“自动化，自动化，自动化。”

——Cay Horstman

手动部署过渡服务器的意义通过自动部署才能体现出来。部署的过程能重复执行，我们才能确信部署到生产环境时不会出错。

使用 Fabric 可以在服务器中自动执行命令。fabric 3 是针对 Python 3 的派生版本：

```
$ pip install fabric3
```

> 安装 fabric3 的过程中，只要最后提示“Successfully installed...”，就可以放心忽略“failed building wheel”错误。

Fabric 的使用方法一般是创建一个名为 fabfile.py 的文件，在这个文件中定义一个或多个函数，然后使用命令行工具 `fab` 调用，就像这样：

```
fab function_name:host=SERVER_ADDRESS
```

这个命令会调用名为 `function_name` 的函数，并传入要连接的服务器地址 `SERVER_ADDRESS`。`fab` 命令还有很多其他参数，可以指定用户名和密码等，详情可执行 `fab --help` 命令查阅。

11.1　分析一个Fabric部署脚本

Fabric 的用法最好通过一个实例讲解。我事先写好了一个脚本[1]，自动执行前一章用到的所有部署步骤。在这个脚本中，主函数是 deploy，我们在命令行中要调用的就是这个函数。deploy 函数还会调用几个辅助函数，我们会在过程中逐一讲解。

deploy_tools/fabfile.py (ch09l001)

```
from fabric.contrib.files import append, exists, sed
from fabric.api import env, local, run
import random

REPO_URL = 'https://github.com/hjwp/book-example.git'  ❶

def deploy():
    site_folder = f'/home/{env.user}/sites/{env.host}'  ❷❸
    source_folder = site_folder + '/source'
    _create_directory_structure_if_necessary(site_folder)
    _get_latest_source(source_folder)
    _update_settings(source_folder, env.host)  ❷
    _update_virtualenv(source_folder)
    _update_static_files(source_folder)
    _update_database(source_folder)
```

❶　要把常量 REPO_URL 的值改成代码分享网站中你仓库的 URL。

❷　env.host 的值是在命令行中指定的服务器地址，例如 superlists.ottg.eu。

❸　env.user 的值是登录服务器时使用的用户名。

希望辅助函数的名字能表明各自的作用。理论上 fabfile.py 中的每个函数都能在命令行中调用，所以我使用了一种约定，凡是以下划线开头的函数都不是 fabfile.py 的“公开 API”。这些辅助函数按照执行的顺序排列。

11.1.1　分析一个Fabric部署脚本

创建目录结构的方法如下，即便某个文件夹已经存在也不会报错：

deploy_tools/fabfile.py (ch09l002)

```
def _create_directory_structure_if_necessary(site_folder):
    for subfolder in ('database', 'static', 'virtualenv', 'source'):
        run(f'mkdir -p {site_folder}/{subfolder}')  ❶❷
```

❶　run 是最常用的 Fabric 函数，作用是在服务器中执行指定的 shell 命令。本章将用 run 命令替代前两章手动执行的多个命令。

注 1：BBC 的儿童节目《蓝彼得》中经常说这句话，因为这是个直播节目，为了节省时间，往往要事先做好所需的道具。这句话的原文是“Here’s one I made earlier”。——译者注

❷ mkdir -p 是 mkdir 的一个有用变种，它有两个优势：其一，深入多个文件夹层级创建目录；其二，只在必要时创建目录。所以，mkdir -p /tmp/foo/bar 除了创建目录 bar 之外，如果需要，还会创建父级目录 foo。而且，如果目录 bar 已经存在，也不会报错。[2]

11.1.2 使用Git拉取源码

接下来，我们想像前面那样使用 git pull 把最新的源码下载到服务器中：

deploy_tools/fabfile.py (ch09l003)

```
def _get_latest_source(source_folder):
    if exists(source_folder + '/.git'):   ❶
        run(f'cd {source_folder} && git fetch')   ❷❸
    else:
        run(f'git clone {REPO_URL} {source_folder}')   ❹
    current_commit = local("git log -n 1 --format=%H", capture=True)   ❺
    run(f'cd {source_folder} && git reset --hard {current_commit}')   ❻
```

❶ exists 检查服务器中是否有指定的文件夹或文件。我们指定的是隐藏文件夹 .git，检查仓库是否已经克隆到文件夹中。

❷ 很多命令都以 cd 开头，其目的是设定当前工作目录。Fabric 没有状态记忆，所以下次运行 run 命令时不知道在哪个目录中。[3]

❸ 在现有仓库中执行 git fetch 命令的作用是从网络中拉取最新提交（与 git pull 类似，但是不会立即更新线上源码）。

❹ 如果仓库不存在，就执行 git clone 命令克隆一份全新的源码。

❺ Fabric 中的 local 函数在本地电脑中执行命令，这个函数其实是对 subprocess.Popen 的再包装，不过用起来十分方便。我们捕获 git log 命令的输出，获取本地仓库中当前提交的 ID。这么做的结果是，服务器中代码将和本地检出的代码版本一致（前提是已经把代码推送到服务器）。

❻ 执行 git reset --hard 命令，切换到指定的提交。这个命令会撤销在服务器中对代码仓库所做的任何改动。

这个函数的最终结果是，全新部署时执行 git clone，已有代码时执行 git fetch 和 git reset --hard。手动部署时，我们执行的是等效的 git pull，但是使用 reset --hard 能强制覆盖本地改动。

> 为了让这个脚本可用，你要执行 git push 命令把本地仓库推送到代码分享网站，这样服务器才能拉取仓库，再执行 git reset 命令。如果你遇到 Could not parse object 错误，可以执行 git push 命令试试。

注 2：如果你想知道构建路径为什么使用 f 字符串而不用前面用过的 os.path.join，我告诉你，因为在 Windows 中运行这个脚本，path.join 会使用反斜线，但在服务器中却要使用斜线。这是一个常见陷阱。

注 3：Fabric 本身也提供了 cd 函数，但我觉得本章要多次用到这个函数，太啰唆。

11.1.3　更新settings.py

然后更新配置文件，设置 `ALLOWED_HOSTS` 和 `DEBUG`，还要创建一个密钥：

deploy_tools/fabfile.py (ch09l004)

```
def _update_settings(source_folder, site_name):
    settings_path = source_folder + '/superlists/settings.py'
    sed(settings_path, "DEBUG = True", "DEBUG = False")   ❶
    sed(settings_path,
        'ALLOWED_HOSTS =.+$',
        f'ALLOWED_HOSTS = ["{site_name}"]'   ❷
    )
    secret_key_file = source_folder + '/superlists/secret_key.py'
    if not exists(secret_key_file):   ❸
        chars = 'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)'
        key = ''.join(random.SystemRandom().choice(chars) for _ in range(50))
        append(secret_key_file, f'SECRET_KEY = "{key}"')
    append(settings_path, '\nfrom .secret_key import SECRET_KEY')   ❹❺
```

❶ Fabric 提供的 `sed` 函数的作用是在文件中替换字符串。这里我们把 `DEBUG` 的值由 `True` 改成 `False`。

❷ 这里使用 `sed` 调整 `ALLOWED_HOSTS` 的值，使用正则表达式匹配正确的代码行。

❸ Django 有几处加密操作要使用 `SECRET_KEY`：cookie 和 CSRF 保护。在服务器中和源码仓库中使用不同的密钥是个好习惯，因为仓库中的代码能被所有人看到。如果还没有密钥，这段代码会生成一个新密钥，然后写入密钥文件。有密钥后，每次部署都要使用相同的密钥。更多信息参见 Django 文档。

❹ `append` 的作用是在文件末尾添加一行内容。（这个函数很聪明，如果要添加的行已经存在，就不会再次添加；但如果文件末尾不是一个空行，它却不能自动添加一个空行。因此我们加上了 `\n`。）

❺ 我使用的是**相对导入**（relative import，使用 `from .secret_key` 而不是 `from secret_key`），目的是确保从本地而不是从 `sys.path` 中其他位置的模块中导入。下一章我会更深入地介绍相对导入。

> 以上是修改服务器配置文件的一种方法，另一种常用的方法是使用环境变量（详情请参见第 21 章）。你可以根据个人喜好选择。

11.1.4　更新虚拟环境

接下来创建或更新虚拟环境：

deploy_tools/fabfile.py (ch09l005)

```
def _update_virtualenv(source_folder):
    virtualenv_folder = source_folder + '/../virtualenv'
    if not exists(virtualenv_folder + '/bin/pip'):   ❶
        run(f'python3.6 -m venv {virtualenv_folder}')
    run(f'{virtualenv_folder}/bin/pip install -r {source_folder}/requirements.txt')   ❷
```

❶ 在 virtualenv 文件夹中查找可执行文件 pip，以此检查虚拟环境是否存在。

❷ 然后和之前一样，执行 pip install -r 命令。

更新静态文件只需要一个命令：

deploy_tools/fabfile.py (ch09l006)

```
def _update_static_files(source_folder):
    run(
        f'cd {source_folder}'   ❶
        ' && ../virtualenv/bin/python manage.py collectstatic --noinput'   ❷
    )
```

❶ 在 Python 中，可以像这样把一行长字符串分为多行，最终拼接为一个字符串。如果真正想要的是字符串列表，但是忘了逗号，就经常会出问题。

❷ 如果需要执行 Django 的 manage.py 命令，就要指定虚拟环境中二进制文件夹，确保使用的是虚拟环境中的 Django 版本，而不是系统中的版本。

11.1.5　需要时迁移数据库

最后，执行 manage.py migrate 命令更新数据库：

deploy_tools/fabfile.py (ch09l007)

```
def _update_database(source_folder):
    run(
        f'cd {source_folder}'
        ' && ../virtualenv/bin/python manage.py migrate --noinput'
    )
```

指定 --noinput 选项的目的是不让 Fabric 难以处理的交互式确认（回答“yes”或“no”）出现。

到此结束！虽然要理解很多新东西，但这是值得的，因为我们把整个手动过程变成自动过程了。而且通过一些逻辑处理，我们既能部署全新的服务器，也能更新现有的服务器。如果你喜欢源自拉丁语的词，可以称这个过程是**幂等**的（idempotent[4]），即不管执行一次还是执行多次，效果是一样的。

11.2　试用部署脚本

下面在现有的过渡网站中试一下这个脚本，看它是如何更新现有网站的：

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu

[superlists-staging.ottg.eu] Executing task 'deploy'
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
```

注 4：“幂等”的英文 idempotent，拉丁语词根是 idem。——译者注

```
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[localhost] local: git log -n 1 --format=%H
[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: HEAD is now at 85a6c87 Add a fabfile for autom
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False
[superlists-staging.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists-staging.ott
[superlists-staging.ottg.eu] run: echo 'SECRET_KEY = '\\''4p2u8fi6)bltep(6nd_3tt
[superlists-staging.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "

[superlists-staging.ottg.eu] run: /home/elspeth/sites/superlists-staging.ottg.eu
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Cleaning up...
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out:
[superlists-staging.ottg.eu] out: 0 static files copied, 11 unmodified.
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: Creating tables ...
[superlists-staging.ottg.eu] out: Installing custom SQL ...
[superlists-staging.ottg.eu] out: Installing indexes ...
[superlists-staging.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists-staging.ottg.eu] out:
Done.
Disconnecting from superlists-staging.ottg.eu... done.
```

太棒了。我喜欢让电脑成页地显示这种输出（事实上，我完全无法阻止自己制造 20 世纪 70 年代的电脑发出的“啵呃 – 啵呃 – 啵呃”声音，就像《异形》中的电脑 Mother 一样[5]）。如果仔细看这些输出，会发现脚本在执行我们的命令：虽然目录结构已经建好，但 `mkdir -p` 命令还是能顺利执行；然后执行 `git pull` 命令，拉取我们刚刚提交的几次改动；`sed` 和 `echo >>` 修改 settings.py 文件；然后顺利执行完 `pip install -r requirements.txt` 命令，注意，现有的虚拟环境中已经安装了全部所需的包；`collectstatic` 命令发现静态文件也收集好了；最后，执行 `migrate` 命令。这个过程完全没障碍。

配置 Fabric

如果使用 SSH 密钥登录，密钥存储在默认的位置，而且本地电脑和服务器使用相同的用户名，那么无须配置即可直接使用 Fabric。如果不满足这几个条件，就要配置用户名、SSH 密钥的位置或密码等，才能让 `fab` 执行命令。

这几个信息可在命令行中传给 Fabric。更多信息可执行 `$ fab --help` 命令查看，或者阅读 Fabric 的文档。

注 5：Mother 是电影《异形》系列中诺史莫号的中央控制系统。——译者注

11.2.1　部署到线上服务器

下面在线上服务器中试试这个脚本：

```
$ fab deploy:host=elspeth@superlists.ottg.eu

$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/databa
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/static
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/virtua
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/source
[superlists.ottg.eu] run: git clone https://github.com/hjwp/book-example.git /ho
[superlists.ottg.eu] out: Cloning into '/home/elspeth/sites/superlists.ottg.eu/s
[superlists.ottg.eu] out: remote: Counting objects: 3128, done.
[superlists.ottg.eu] out: Receiving objects:   0% (1/3128)
[...]
[superlists.ottg.eu] out: Receiving objects: 100% (3128/3128), 2.60 MiB | 829 Ki
[superlists.ottg.eu] out: Resolving deltas: 100% (1545/1545), done.
[superlists.ottg.eu] out:

[localhost] local: git log -n 1 --format=%H
[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && gi
[superlists.ottg.eu] out: HEAD is now at 6c8615b use a secret key file
[superlists.ottg.eu] out:
[superlists.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False/g' "$(e
[superlists.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists.ottg.eu"]' >> "$(ec
[superlists.ottg.eu] run: echo 'SECRET_KEY = '\\''mqu(ffwid5vleol%ke^jil*x1mkj-4
[superlists.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "$(echo /
[superlists.ottg.eu] run: python3.6 -m venv /home/elspeth/sites/superl
[superlists.ottg.eu] out: Using interpreter /usr/bin/python3.6
[superlists.ottg.eu] out: Using base prefix '/usr'
[superlists.ottg.eu] out: New python executable in /home/elspeth/sites/superlist
[superlists.ottg.eu] out: Also creating executable in /home/elspeth/sites/superl
[superlists.ottg.eu] out: Installing Setuptools............................done.

[superlists.ottg.eu] out: Installing Pip...................................done.
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: /home/elspeth/sites/superlists.ottg.eu/source/../virtu
[superlists.ottg.eu] out: Downloading/unpacking Django==1.11 (from -r /home/el
[superlists.ottg.eu] out:   Downloading Django-1.11.tar.gz (8.0MB):
[...]
[superlists.ottg.eu] out:   Downloading Django-1.11.tar.gz (8.0MB): 100% 8.0M
[superlists.ottg.eu] out:   Running setup.py egg_info for package Django
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     warning: no previously-included files matching '__
[superlists.ottg.eu] out:     warning: no previously-included files matching '*.
[superlists.ottg.eu] out: Downloading/unpacking gunicorn==17.5 (from -r /home/el
[superlists.ottg.eu] out:   Downloading gunicorn-17.5.tar.gz (367kB): 100%  367k
[...]

[superlists.ottg.eu] out:   Downloading gunicorn-17.5.tar.gz (367kB): 367kB down
[superlists.ottg.eu] out:   Running setup.py egg_info for package gunicorn
```

```
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: Installing collected packages: Django, gunicorn
[superlists.ottg.eu] out:   Running setup.py install for Django
[superlists.ottg.eu] out:     changing mode of build/scripts-3.3/django-admin.py
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     warning: no previously-included files matching '__
[superlists.ottg.eu] out:     warning: no previously-included files matching '*.
[superlists.ottg.eu] out:     changing mode of /home/elspeth/sites/superlists.ot
[superlists.ottg.eu] out:   Running setup.py install for gunicorn
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     Installing gunicorn_paster script to /home/elspeth
[superlists.ottg.eu] out:     Installing gunicorn script to /home/elspeth/sites/
[superlists.ottg.eu] out:     Installing gunicorn_django script to /home/elspeth
[superlists.ottg.eu] out: Successfully installed Django gunicorn
[superlists.ottg.eu] out: Cleaning up...
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && ..
[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source
[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source
[...]
[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: 11 static files copied.
[superlists.ottg.eu] out:
[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && ..
[superlists.ottg.eu] out: Creating tables ...
[superlists.ottg.eu] out: Creating table auth_permission
[...]
[superlists.ottg.eu] out: Creating table lists_item
[superlists.ottg.eu] out: Installing custom SQL ...
[superlists.ottg.eu] out: Installing indexes ...
[superlists.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists.ottg.eu] out:
Done.
Disconnecting from superlists.ottg.eu... done.
```

啵呃－啵呃－啵呃。可以看出，这个脚本执行的路径有点不同。这一次，执行 `git clone` 命令克隆一个全新的仓库，而没有执行 `git pull`；而且从零开始创建了一个新的虚拟环境，还安装了 pip 和 Django；`collectstatic` 命令这次真的创建了很多新文件；`migrate` 看起来也完成了任务。

11.2.2　使用sed配置Nginx和Gunicorn

把网站放到生产环境之前还要做什么呢？根据配置笔记，还要使用模板文件创建 Nginx 虚拟主机和 Systemd 服务。使用 Unix 命令行工具完成这一操作怎么样？

```
elspeth@server:$ sed "s/SITENAME/superlists.ottg.eu/g" \
    source/deploy_tools/nginx.template.conf \
    | sudo tee /etc/nginx/sites-available/superlists.ottg.eu
```

`sed`（stream editor，流编辑器）的作用是编辑文本流。Fabric 中进行文本替换的函数也叫 `sed`，这并不是巧合。这里，使用 `s/replaceme/withthis/g` 句法把字符串 `SITENAME` 替换成

网站的地址。[6] 然后使用管道操作（|）把文本流传给一个有 root 权限的用户处理（sudo），把传入的文本流写入一个文件，即 sites-available 文件夹中的一个虚拟主机配置文件。

然后使用一个符号链激活这个文件配置的虚拟主机：

```
elspeth@server:$ sudo ln -s ../sites-available/superlists.ottg.eu \
    /etc/nginx/sites-enabled/superlists.ottg.eu
```

再使用 sed 编写 Systemd 服务：

```
elspeth@server: sed "s/SITENAME/superlists.ottg.eu/g" \
    source/deploy_tools/gunicorn-systemd.template.service \
    | sudo tee /etc/systemd/system/gunicorn-superlists.ottg.eu.service
```

最后，启动这两个服务：

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ sudo systemctl enable gunicorn-superlists.ottg.eu
elspeth@server:$ sudo systemctl start gunicorn-superlists.ottg.eu
```

现在访问网站，见图 11-1。运行起来了，太棒了！

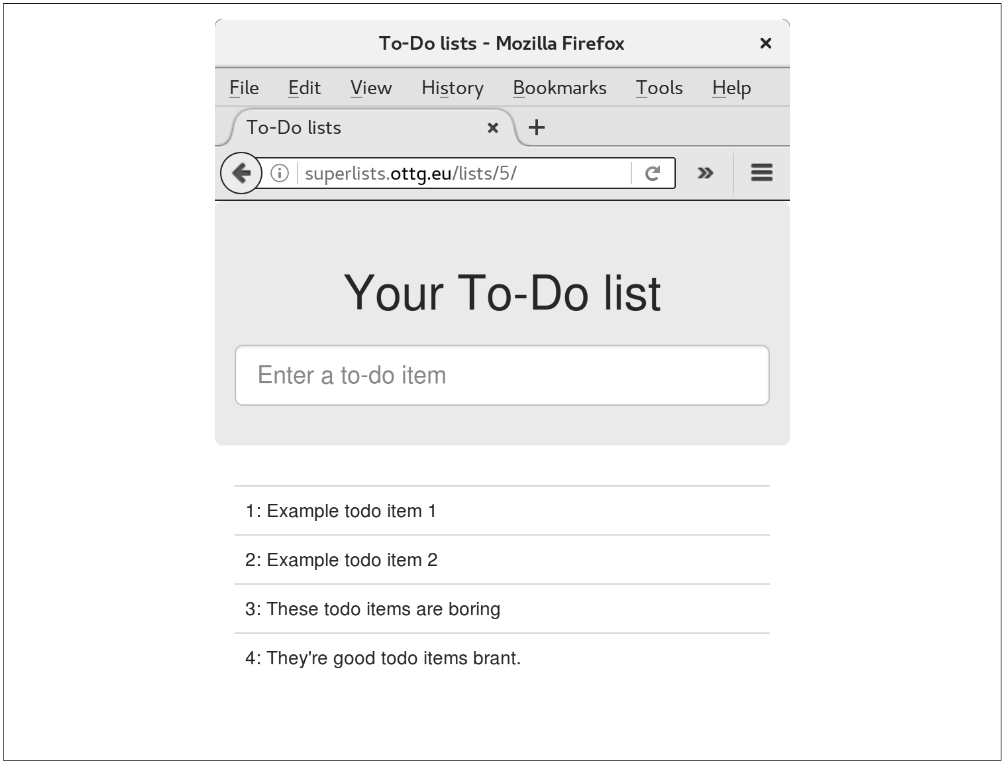


图 11-1：啵呃－啵呃－啵呃……网站运行起来了

注 6：你可能在网上见过电脑达人使用奇怪的 s/change-this/to-this 的写法，这下你知道它的作用了吧。

做得不错。可靠的 fabfile，赏你一块饼干。现在可以把它添加到仓库中了：

```
$ git add deploy_tools/fabfile.py
$ git commit -m "Add a fabfile for automated deploys"
```

11.3　使用Git标签标注发布状态

最后还要做些管理操作。为了保留历史标记，使用 Git 标签（tag）标注代码库的状态，指明服务器中当前使用的是哪个版本：

```
$ git tag LIVE
$ export TAG=$(date +DEPLOYED-%F/%H%M) # 生成一个时间戳
$ echo $TAG # 会显示“DEPLOYED-”和时间戳
$ git tag $TAG
$ git push origin LIVE $TAG # 推送标签
```

现在，无论何时都能轻松查看当前代码库和服务器中的版本有何差异。这个操作在后面介绍数据库迁移的章节中也会用到。看一下提交历史中的标签：

```
$ git log --graph --oneline --decorate

[...]
```

总之，现在我们部署了一个线上网站。告诉你的朋友吧！如果他们不感兴趣，就告诉自己的妈妈！下一章我们要继续编程。

11.4　延伸阅读

部署没有唯一正确的方法，而且我无论如何也算不上资深专家。我试着把你领进门，但有很多事情可以使用不同的方法处理，还有很多很多的知识要学习。下面我列出了一些阅读材料，仅供参考。

- Hynek Schlawack 的文章“Solid Python Deployments for Everybody”。
- Dan Bravender 的文章“Git-based fabric deploys are awesome”。
- Dan Greenfield 和 Audrey Roy 合著的 *Two Scoops of Django* 中与部署相关的章节。
- Heroku 团队写的“The 12-factor App”。

如果想知道如何自动完成配置过程，以及如何使用 Fabric 的替代品 Ansible，请阅读附录 C。

自动部署

- Fabric

  Fabric 允许在 Python 脚本中编写可在服务器中执行的命令。这个工具很适合自动执行服务器管理任务。

- 幂等

  如果部署脚本要在已经配置的服务器中运行，就要把它设计成**既可**在全新的服务器中运行，**又能**在已经配置的服务器中运行。

- 把配置文件纳入版本控制

  一定不能只在服务器中保存一份配置文件副本。配置文件对应用非常重要，应该和其他文件一样纳入版本控制。

- 自动配置

  最终，**所有**操作都要实现自动化，包括配置全新的服务器和安装所需的全部正确软件。配置的过程中会和主机供应商的 API 交互。

- 配置管理工具

  Fabric 很灵活，但其逻辑还是基于脚本的。高级工具使用声明式的方法，用起来更方便。Ansible 和 Vagrant 都值得一试（参见附录 C），此外还有很多同类工具，例如 Chef、Puppet、Salt 和 Juju 等。

第 12 章

输入验证和测试的组织方式

接下来要实现的功能是输入验证。测试越写越多，慢慢地你会发现，把所有测试都写在 functional_tests.py 和 tests.py 中有诸多不便，因此我们将重新组织测试，将它们写入多个文件——这算得上是对测试本身的重构。

此外，我们还将定义一个通用的显式等待辅助方法。

12.1 针对验证的功能测试：避免提交空待办事项

我们的网站开始有用户了。我们注意到用户有时会犯错，把他们的清单弄得一团糟，例如不小心提交空的待办事项，或者在一个清单中输入两个相同的待办事项。计算机能帮助我们避免犯这种愚蠢的错误，看一下能否让网站提供这种帮助。

下面是一个功能测试的大纲：

functional_tests/tests.py (ch11l001)

```
def test_cannot_add_empty_list_items(self):
    # 伊迪丝访问首页，不小心提交了一个空待办事项
    # 输入框中没输入内容，她就按下了回车键

    # 首页刷新了，显示一个错误消息
    # 提示待办事项不能为空

    # 她输入一些文字，然后再次提交，这次没问题了

    # 她有点儿调皮，又提交了一个空待办事项

    # 在清单页面她看到了一个类似的错误消息
```

```
        # 输入文字之后就没问题了
        self.fail('write me!')
```

测试写得很好，但功能测试文件变得有点儿臃肿，在继续之前，要把功能测试分成多个文件，每个文件中只放一个测试方法。

还记得吗？功能测试和“用户故事”联系紧密。如果使用问题跟踪程序等项目管理工具，你可能想让每个文件对应一个问题或工单（ticket），而且文件名中要包含工单的编号。如果你喜欢使用“功能”的概念考虑问题（一个功能可能包含多个用户故事），可以用一个文件对应一个功能，一个文件中只写一个测试类，每个用户故事使用多个测试方法实现。

还要编写一个测试基类，让所有测试类都继承这个基类。下面分步介绍分拆过程。

12.1.1　跳过测试

重构时最好能让整个测试组件都通过。刚才我们故意编写了一个失败测试，现在要使用 `unittest` 提供的修饰器 `@skip` 临时禁止执行这个测试方法：

functional_tests/tests.py (ch11l001-1)

```
from unittest import skip
[...]

    @skip
    def test_cannot_add_empty_list_items(self):
```

这个修饰器告诉测试运行程序，忽略这个测试。再次运行功能测试就会看到这么做起作用了，因为测试组件仍能通过：

```
$ python manage.py test functional_tests
[...]
Ran 4 tests in 11.577s
OK
```

跳过测试很危险，把改动提交到仓库之前记得删掉 `@skip` 修饰器。这就是逐行审查差异的目的。

别忘了“遇红 / 变绿 / 重构”中的“重构”

TDD 有时会被批评说得到的代码架构不好，因为开发者关注的是怎么让测试通过，没有停下来思考整个系统应该怎么设计。我觉得这么说有点不公平。

TDD 不是万能良药。你仍要花时间考虑好的设计。不过开发者经常会忘记“遇红 / 变绿 / 重构”中还有“重构”这一步。使用 TDD，为了让测试通过，可以随意丢掉旧代码，但它**也**要求你在测试通过之后花点儿时间重构，改进设计。否则“技术债务”将高高筑起。

不过，重构的最佳方法往往不那么容易想到，可能等到写下代码之后的几天、几周甚至几个月，处理完全无关的事情时，突然灵光一闪才能想出来。在解决其他问题的途中，应该停下来去重构以前的代码吗？

这要视情况而定。比如像本章开始这种情况，我们还没有开始编写新代码，知道一切都能正常运行，所以可以跳过刚编写的功能测试（让测试全部通过），先重构。

在本章后面的内容中还会遇到需要重构的代码。届时，我们不能冒险在无法正常运行的应用中重构，可以在便签上做个记录，等测试组件能全部通过之后再重构。

12.1.2　把功能测试分拆到多个文件中

先把各个测试方法放在单独的类中，但仍然保存在同一个文件里：

functional_tests/tests.py (ch11l002)

```
class FunctionalTest(StaticLiveServerTestCase):

    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def wait_for_row_in_list_table(self, row_text):
        [...]


class NewVisitorTest(FunctionalTest):

    def test_can_start_a_list_for_one_user(self):
        [...]
    def test_multiple_users_can_start_lists_at_different_urls(self):
        [...]


class LayoutAndStylingTest(FunctionalTest):

    def test_layout_and_styling(self):
        [...]



class ItemValidationTest(FunctionalTest):

    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

然后运行功能测试，看是否仍能通过：

```
Ran 4 tests in 11.577s

OK
```

这么做可能有点劳动量，或许能找到一种步骤更少的方法。但是，正如我一直所说的，针对简单的情况练习步步为营的方法，以后遇到复杂的情况就能游刃有余。

现在分拆这个测试文件，一个类写入一个文件，而且还有一个文件用来保存所有测试类都继承的基类。要复制四份 test.py，重命名各个文件，然后删除各文件中不需要的代码：

```
$ git mv functional_tests/tests.py functional_tests/base.py
$ cp functional_tests/base.py functional_tests/test_simple_list_creation.py
$ cp functional_tests/base.py functional_tests/test_layout_and_styling.py
$ cp functional_tests/base.py functional_tests/test_list_item_validation.py
```

base.py 只需保留 `FunctionalTest` 类，其他代码全部删掉。留下基类中的辅助方法，因为觉得在新的功能测试中会用到：

functional_tests/base.py (ch11l003)

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium import webdriver
from selenium.common.exceptions import WebDriverException
import time

MAX_WAIT = 10



class FunctionalTest(StaticLiveServerTestCase):

    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def wait_for_row_in_list_table(self, row_text):
        [...]
```

把辅助方法放在 `FunctionalTest` 基类中是避免功能测试代码重复的方法之一。第 25 章会用到“页面模式”（page pattern），与这种方法有关，但不用继承，用组合模式。

我们编写的第一个功能测试现在放在单独的文件中了，而且这个文件中只有一个类和一个测试方法：

functional_tests/test_simple_list_creation.py (ch11l004)

```
from .base import FunctionalTest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys


class NewVisitorTest(FunctionalTest):

    def test_can_start_a_list_for_one_user(self):
        [...]
```

```
    def test_multiple_users_can_start_lists_at_different_urls(self):
        [...]
```

我用到了相对导入（from .base），有些人喜欢在 Django 应用中大量使用这种导入方式（例如，视图可能会使用 from .models import List 导入模型，而不用 from list.models）。这其实是个人喜好问题，只有十分确定要导入的文件位置不会变化时，我才会选择使用相对导入。这里使用相对导入是因为，我确定所有测试文件都会和它们要继承的 base.py 放在一起。

针对布局和样式的功能测试现在也放在独立的文件和类中：

functional_tests/test_layout_and_styling.py (ch11l005)

```
from selenium.webdriver.common.keys import Keys
from .base import FunctionalTest


class LayoutAndStylingTest(FunctionalTest):
        [...]
```

刚编写的验证测试也放在单独的文件中了：

functional_tests/test_list_item_validation.py (ch11l006)

```
from selenium.webdriver.common.keys import Keys
from unittest import skip
from .base import FunctionalTest


class ItemValidationTest(FunctionalTest):

    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

可以再次执行 manage.py test functional_tests 命令，确保一切都正常，还要确认所有三个测试都运行了：

```
Ran 4 tests in 11.577s

OK
```

现在可以删掉 @skip 修饰器了：

functional_tests/test_list_item_validation.py (ch11l007)

```
class ItemValidationTest(FunctionalTest):

    def test_cannot_add_empty_list_items(self):
        [...]
```

12.1.3　运行单个测试文件

拆分之后有个附带的好处——可以运行单个测试文件，如下所示：

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
AssertionError: write me!
```

太好了，如果只关心其中一个测试，现在无须坐等所有功能测试都运行完就能看到结果了。不过，记得要不时运行所有功能测试，检查是否有回归。本书后面的内容会介绍如何把这项任务交给自动化持续集成循环完成。现在，先提交：

```
$ git status
$ git add functional_tests
$ git commit -m "Moved Fts into their own individual files"
```

很好，我们把功能测试拆分到不同的文件中了。接下来，我们将着手编写功能测试。但在此之前，你可能猜到了，我们还需拆分单元测试文件。

12.2 功能测试新工具：通用显式等待辅助方法

现在开始实现本章开头编写的测试，至少先把前面的部分写好：

functional_tests/test_list_item_validation.py (ch11l008)

```
def test_cannot_add_empty_list_items(self):
    # 伊迪丝访问首页，不小心提交了一个空待办事项
    # 输入框中没输入内容，她就按下了回车键
    self.browser.get(self.live_server_url)
    self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)

    # 首页刷新了，显示一个错误消息
    # 提示待办事项不能为空
    self.assertEqual(
        self.browser.find_element_by_css_selector('.has-error').text,   ❶
        "You can't have an empty list item"   ❷
    )

    # 她输入一些文字，然后再次提交，这次没问题了
    self.fail('finish this test!')
    [...]
```

这可想得太天真了。

❶ 通过 CSS 类 `.has-error` 查找错误文本。Bootstrap 为错误文本提供了很多有用的样式。

❷ 确认错误文本中有我们想显示的消息。

不过，你能看出这个测试有什么潜在问题吗？

好吧，本节的标题已经给出提示：如果页面刷新，就要显式等待；否则，Selenium 可能会在页面加载之前查找 `.has-error` 元素。

> 只要通过 `Keys.ENTER` 或点击按钮提交表单后页面要刷新，在下一个断言之前可能就需要显式等待。

首次需要显式等待时，我们定义了一个辅助方法。但对这个测试来说，你可能觉得用辅助方法有点小题大做。不过，在测试中能使用通用的方式表达“等到断言通过”也不错，比如这样：

functional_tests/test_list_item_validation.py (ch11l009)

```
    [...]
        # 首页刷新了，显示了一条错误消息
        # 提示待办事项不能为空
        self.wait_for(lambda: self.assertEqual(  ❶
            self.browser.find_element_by_css_selector('.has-error').text,
            "You can't have an empty list item"
        ))
```

❶ 不再直接调用断言，而是把断言包装到一个 `lambda` 函数中，然后再把它传给一个打算命名为 `wait_for` 的辅助方法。

> 如果你从未在 Python 中见过 `lambda` 函数，请阅读后文“`lambda` 函数”。

那应该如何实现这个 `wait_for` 方法呢？打开 base.py，复制现有的 `wait_for_row_in_list_table` 方法，稍微修改一下：

functional_tests/base.py (ch11l010)

```
    def wait_for(self, fn):  ❶
        start_time = time.time()
        while True:
            try:
                table = self.browser.find_element_by_id('id_list_table')  ❷
                rows = table.find_elements_by_tag_name('tr')
                self.assertIn(row_text, [row.text for row in rows])
                return
            except (AssertionError, WebDriverException) as e:
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)
```

❶ 复制现有的方法，但将其重命名为 `wait_for` 并修改参数，期待传入一个函数。

❷ 现在的代码仍然检查表格中的行。怎样修改才能支持传入的 `fn` 函数呢？

像这样：

functional_tests/base.py (ch11l011)

```
    def wait_for(self, fn):
        start_time = time.time()
        while True:
            try:
                return fn()  ❶
```

```
            except (AssertionError, WebDriverException) as e:
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)
```

❶ try/except 的主体不再检查表格中的行，而是调用传入的函数。如果没有异常抛出，就返回传入的那个函数的返回值，立即跳出循环。

lambda 函数

在 Python 中，一次性单行函数使用 lambda 构建，这样免去了使用 def..(): 和缩进代码块的麻烦。

```
>>> myfn = lambda x: x+1
>>> myfn(2)
3
>>> myfn(5)
6
>>> adder = lambda x, y: x + y
>>> adder(3, 2)
5
```

在下述示例中，我们把一段本应立即执行的代码定义为一个函数，然后作为参数传给 lambda 函数，留待以后执行，而且可以多次执行：

```
>>> def addthree(x):
...     return x + 3
...
>>> addthree(2)
5
>>> myfn = lambda: addthree(2)  # 注意，这里的addthree不会立即调用
>>> myfn
<function <lambda> at 0x7f3b140339d8>
>>> myfn()
5
>>> myfn()
5
```

下面来看 wait_for 辅助方法的实际效果：

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
======================================================================
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_list_item_validation.py", line
15, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual(  ❶
  File "/.../superlists/functional_tests/base.py", line 37, in wait_for
    raise e  ❷
  File "/.../superlists/functional_tests/base.py", line 34, in wait_for
    return fn()  ❷
  File "/.../superlists/functional_tests/test_list_item_validation.py", line
```

```
16, in <lambda>  ❸
    self.browser.find_element_by_css_selector('.has-error').text,  ❸
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
 ---------------------------------------------------------------------
Ran 1 test in 10.575s

FAILED (errors=1)
```

调用跟踪的顺序有点儿乱，不过我们或多或少能猜到到底发生了什么。

❶ 功能测试的第 15 行进入 `self.wait_for` 辅助方法，传入 `assertEqual` 的 lambda 版本。

❷ 进入 base.py 中的 `self.wait_for`，可以看到我们调用了那个 `lambda` 函数，但是由于超时而抛出 `raise e`。

❸ 为了查明异常来自何处，调用跟踪又把我们带到 test_list_item_validation.py 文件中的 `lambda` 函数主体里，并且告诉我们，是在尝试查找 `.has-error` 元素时失败的。

现在我们进入了函数式编程领域，即把一个函数作为参数传给另一个函数——这可能有点烧脑，我当初也是历经一番坎坷才理解的。多读几遍代码和功能测试，慢慢体会。如果还是不能理解，也别烦忧，在使用的过程中慢慢领会。本书将多次使用函数式编程，你会领略它的强大之处的。

12.3　补完功能测试

把功能测试补完：

functional_tests/test_list_item_validation.py (ch11l012)

```
        # 首页刷新了，显示一条错误消息
        # 提示待办事项不能为空
        self.wait_for(lambda: self.assertEqual(
            self.browser.find_element_by_css_selector('.has-error').text,
            "You can't have an empty list item"
        ))

        # 她输入一些文字，然后再次提交，这次没问题了
        self.browser.find_element_by_id('id_new_item').send_keys('Buy milk')
        self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table('1: Buy milk')

        # 她有点儿调皮，又提交了一个空待办事项
        self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)

        # 她在列表页面看到了一条类似的错误消息
        self.wait_for(lambda: self.assertEqual(
            self.browser.find_element_by_css_selector('.has-error').text,
            "You can't have an empty list item"
        ))
```

```
# 输入文字之后就没问题了
self.browser.find_element_by_id('id_new_item').send_keys('Make tea')
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')
self.wait_for_row_in_list_table('2: Make tea')
```

功能测试中的辅助方法

至此，我们定义了两个辅助方法：self.wait_for 和 wait_for_row_in_list_table。前者是通用的，任何功能测试都可能需要等待。

后者还有助于避免功能测试出现重复的代码。如果有一天我们决定改变清单表格的实现，只需修改一个地方，而不用在功能测试中四处修改。

第 25 章和附录 E 还将深入讨论如何合理规划功能测试。

这次提交留给你自己完成，这是你第一次独立提交功能测试。

12.4 重构单元测试，分拆成多个文件

最终开始编写代码前，要为模型编写一个新测试，但在此之前，先要使用类似于功能测试的整理方法整理单元测试。

但这一次有所不同，因为 lists 应用中既有应用代码也有测试代码，所以要把测试放到单独的文件夹中：

```
$ mkdir lists/tests
$ touch lists/tests/__init__.py
$ git mv lists/tests.py lists/tests/test_all.py
$ git status
$ git add lists/tests
$ python manage.py test lists
[...]
Ran 9 tests in 0.034s

OK
$ git commit -m "Move unit tests into a folder with single file"
```

如果测试的输出显示“Ran 0 tests”，有可能是因为你忘了创建 __init__.py 文件——必须有这个文件，否则测试所在的文件夹就不是有效的 Python 包。

现在把 test_all.py 分成两个文件：一个名为 test_views.py，只包含视图测试；另一个名为 test_models.py。先复制两份：

```
$ git mv lists/tests/test_all.py lists/tests/test_views.py
$ cp lists/tests/test_views.py lists/tests/test_models.py
```

然后清理 test_models.py，只留下一个测试方法，所以导入的模块也更少了：

lists/tests/test_models.py (ch11l016)

```
from django.test import TestCase
from lists.models import Item, List


class ListAndItemModelsTest(TestCase):
        [...]
```

test_views.py 只减少了一个类：

lists/tests/test_views.py (ch11l017)

```
--- a/lists/tests/test_views.py
+++ b/lists/tests/test_views.py
@@ -103,34 +104,3 @@ class ListViewTest(TestCase):
         self.assertNotContains(response, 'other list item 1')
         self.assertNotContains(response, 'other list item 2')

-
-
-class ListAndItemModelsTest(TestCase):
-
-    def test_saving_and_retrieving_items(self):
[...]
```

再次运行测试，确保一切正常：

```
$ python manage.py test lists
[...]
Ran 9 tests in 0.040s

OK
```

很好！

```
$ git add lists/tests
$ git commit -m "Split out unit tests into two files"
```

> 有些人喜欢在项目一开始就把单元测试放在一个测试文件夹中。这种做法很棒。我只是想等到必要时再这么做，以免第一章内容过杂。

至此，功能测试和单元测试的组织方式更合理了。下一章将探讨验证规则。

关于组织测试和重构的小贴士

- 把测试放在单独的文件夹中

  就像使用多个文件保存应用代码一样，你也应该把测试放到多个文件中。

  - 对功能测试来说，按照特定功能或用户故事的方式组织。
  - 对单元测试来说，使用一个名为 tests 的文件夹，并在其中添加 __init__.py 文件。
  - 或许可以把针对一个源码文件的测试放在一个单独的文件中。在 Django 中，往往有 test_models.py、test_views.py 和 test_forms.py。
  - 每个函数和类都至少有一个占位测试。

- 别忘了“遇红/变绿/重构”中的“重构”

  编写测试的主要目的是让你重构代码！一定要重构，尽量让代码（包括测试）变得简洁。

- 测试失败时别重构

  - 一般情况下如此。
  - 不算正在处理的功能测试。
  - 如果测试的对象还没实现，可以先为测试方法加上 `@skip` 装饰器。
  - 更常见的做法是：记下想重构的地方，完成手头上的活儿，等应用处于正常状态时再重构。
  - 提交代码之前别忘了删掉所有 `@skip` 装饰器！一定要逐行审查差异，找出需要删除的每一个地方。

- 尝试通用的`wait_for`辅助方法

  使用专门的辅助方法实现显式等待是个不错的主意，能让测试更易于阅读，但有时单行断言或 Selenium 交互也需要等待一段时间。`self.wait_for` 很符合我的需求，不过你可能需要稍微不同的方式。

第13章

数据库层验证

接下来的几章将实现并测试用户输入验证。

这里有相当一部分内容是专门针对 Django 的，很少涉及 TDD 原理。但这并不意味着你学不到关于测试的新知识——其实我们将讨论很多有趣的测试小知识点，但将更关注如何适应测试、如何跟上 TDD 的节奏以及如何完成手上的工作。

这三章都很短，学完之后，我们将稍微接触一下 JavaScript，然后结束第二部分。第三部分将深入讨论 TDD 方法论的细节，比如比较单元测试和整合测试、介绍模拟技术等。敬请期待！

不过现在要稍微讨论一下验证。先运行功能测试，看一下目前进展到哪里了：

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
======================================================================
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_list_item_validation.py", line
15, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual(
[...]
  File "/.../superlists/functional_tests/test_list_item_validation.py", line

16, in <lambda>
    self.browser.find_element_by_css_selector('.has-error').text,
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
```

功能测试指明，用户输入空待办事项时，期望看到一个错误消息。

13.1　模型层验证

Web 应用的验证放在两个地方：客户端（稍后会看到，使用的是 JavaScript 或 HTML5 属性）和服务器端。在服务器端验证更安全，因为一旦有漏洞或缺陷，客户端验证可以轻易绕过。

在服务器端，尤其是对 Django 而言，也有两个地方可以执行验证：一个是模型层；一个是表单层，位置较高。只要可行，我就会使用低层验证，一方面因为我喜欢数据库和数据库完整性规则，另一方面因为在这一层执行验证更安全——你有时会忘记使用了哪个表单验证输入，但使用的数据库不会变。

13.1.1　self.assertRaises上下文管理器

下面在模型层编写一个单元测试。在 `ListAndItemModelsTest` 中添加一个新测试方法，尝试创建一个空待办事项。这个测试与以往不同，我们要测试的是代码能抛出异常：

lists/tests/test_models.py (ch11l018)

```
from django.core.exceptions import ValidationError
[...]

class ListAndItemModelsTest(TestCase):
    [...]

    def test_cannot_save_empty_list_items(self):
        list_ = List.objects.create()
        item = Item(list=list_, text='')
        with self.assertRaises(ValidationError):
           item.save()
```



如果刚接触 Python，可能从未见过 `with` 语句。它结合“上下文管理器”一起使用，包装一段代码，这些代码的作用往往是设置、清理或处理错误。Python 2.5 的发布说明中有很好的解说。

这是一个新的单元测试技术：如果想检查做某件事是否会抛出异常，可以使用 `self.assertRaises` 上下文管理器。此外还可写成：

```
try:
    item.save()
    self.fail('The save should have raised an exception')
except ValidationError:
    pass
```

不过使用 `with` 语句更简洁。现在运行测试，得到预期失败：

```
    item.save()
AssertionError: ValidationError not raised
```

13.1.2　Django怪异的表现：保存时不验证数据

我们遇到了 Django 的一个怪异表现。**测试本来应该通过的**。阅读 Django 模型字段的文档之后，你会发现 `TextField` 的默认设置是 `blank=False`，也就是说文本字段**应该**拒绝空值。

但为什么测试失败了呢？由于稍微有违常理的历史原因，保存数据时 Django 的模型不会运行全部验证。稍后我们会看到，在数据库中实现的约束，保存数据时都会抛出异常，但 SQLite 不支持文本字段上的强制空值约束，所以我们调用 `save` 方法时无效值悄无声息地通过了验证。

有种方法可以检查约束是否会在数据库层执行：如果在数据库层制定约束，需要执行迁移才能应用约束。但是，Django 知道 SQLite 不支持这种约束，所以如果运行 `makemigrations`，会看到消息说没事可做：

```
$ python manage.py makemigrations
No changes detected
```

不过，Django 提供了一个方法用于运行全部验证，即 `full_clean`。下面我们把这个方法加入测试，看看是否有用：

lists/tests/test_models.py

```
        with self.assertRaises(ValidationError):
            item.save()
            item.full_clean()
```

加入之后，测试就通过了：

```
OK
```

很好！我们通过这个怪异的表现学到了一些 Django 的验证知识。如果忘了需求，把 `text` 字段的约束条件设为 `blank=True`（试一下吧），测试可以提醒我们。

13.2　在视图中显示模型验证错误

下面尝试在视图中处理模型验证，并把验证错误传入模板，让用户看到。在 HTML 中有选择地显示错误可以使用这种方法——检查是否有错误变量传入模板，如果有就在表单下方显示出来：

lists/templates/base.html (ch11l020)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
  <input name="item_text" id="id_new_item"
         class="form-control input-lg"
         placeholder="Enter a to-do item" />
  {% csrf_token %}
  {% if error %}
    <div class="form-group has-error">
      <span class="help-block">{{ error }}</span>
    </div>
  {% endif %}
</form>
```

关于表单控件的更多信息请阅读 Bootstrap 文档。

把错误传入模板是视图函数的任务。看一下 `NewListTest` 类中的单元测试，这里我要使用两种稍微不同的错误处理模式。

在第一种情况中，新建清单视图有可能渲染首页所用的模板，而且还会显示错误消息。单元测试如下：

lists/tests/test_views.py (ch11l021)

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = "You can't have an empty list item"
        self.assertContains(response, expected_error)
```

编写这个测试时，我们手动输入了字符串形式的地址 /lists/new，你可能有点反感。在此之前，我们已经在测试、视图和模板中硬编码了多个地址，这么做有违 DRY 原则。我并不介意测试中有少量重复，但视图和模板中硬编码的地址要引起重视，请在便签上做个记录，稍后重构这些地址。我们不会立即开始重构，因为现在应用无法正常运行，得先让应用回到可运行的状态。

再看测试。现在测试无法通过，因为现在视图返回 302 重定向，而不是正常的 200 响应：

```
AssertionError: 302 != 200
```

我们在视图中调用 `full_clean()` 试试：

lists/views.py

```
def new_list(request):
    list_ = List.objects.create()
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    item.full_clean()
    return redirect(f'/lists/{list_.id}/')
```

看到这个视图的代码，我们找到了一种避免硬编码 URL 的好方法。把这件事记在便签上：



现在模型验证会抛出异常，并且传到了视图中：

```
[...]
  File "/.../superlists/lists/views.py", line 11, in new_list
    item.full_clean()
[...]
django.core.exceptions.ValidationError: {'text': ['This field cannot be
blank.']}
```

下面使用第一种错误处理方案：使用 `try/except` 检测错误。遵从测试山羊的教诲，只加入 `try/except`，其他代码都不动。测试会告诉我们下一步要编写什么代码：

lists/views.py (ch11l025)

```
from django.core.exceptions import ValidationError
[...]

def new_list(request):
    list_ = List.objects.create()
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
    except ValidationError:
        pass
    return redirect(f'/lists/{list_.id}/')
```

加入 `try/except` 之后，测试结果又变成了 302 != 200 错误：

```
AssertionError: 302 != 200
```

下面把 `pass` 改成渲染模板，这么改还兼具检查模板的功能：

lists/views.py (ch11l026)

```
    except ValidationError:
        return render(request, 'home.html')
```

现在测试告诉我们，要把错误消息写入模板：

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list
item' in response
```

为此，可以传入一个新的模板变量：

lists/views.py (ch11l027)

```
    except ValidationError:
        error = "You can't have an empty list item"
        return render(request, 'home.html', {"error": error})
```

不过，看样子没什么用：

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list
item' in response
```

让视图输出一些信息以便调试：

lists/tests/test_views.py

```
expected_error = "You can't have an empty list item"
print(response.content.decode())
self.assertContains(response, expected_error)
```

从输出的信息中我们得知，失败的原因是 Django 转义了 HTML 中的单引号：

```
[...]
<span class="help-block">You can&#39;t have an empty list
item</span>
```

可以在测试中写入：

```
    expected_error = "You can&#39;t have an empty list item"
```

但使用 Django 提供的辅助函数或许是更好的方法：

lists/tests/test_views.py (ch11l029)

```
from django.utils.html import escape
[...]

        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

测试通过了：

```
Ran 11 tests in 0.047s

OK
```

确保无效的输入值不会存入数据库

继续做其他事之前，不知你是否注意到了我们的实现有点逻辑错误？现在即使验证失败仍会创建对象：

lists/views.py

```
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
    except ValidationError:
        [...]
```

要添加一个新单元测试，确保不会保存空待办事项：

lists/tests/test_views.py (ch11l030-1)

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        [...]

    def test_invalid_list_items_arent_saved(self):
```

```
        self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(List.objects.count(), 0)
        self.assertEqual(Item.objects.count(), 0)
```

测试的结果是：

```
[...]
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_views.py", line 40, in
test_invalid_list_items_arent_saved
    self.assertEqual(List.objects.count(), 0)
AssertionError: 1 != 0
```

修正的方法如下：

lists/views.py (ch11l030-2)

```
def new_list(request):
    list_ = List.objects.create()
    item = Item(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
        item.save()
    except ValidationError:
        list_.delete()
        error = "You can't have an empty list item"
        return render(request, 'home.html', {"error": error})
    return redirect(f'/lists/{list_.id}/')
```

功能测试能通过吗？

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
File "/.../superlists/functional_tests/test_list_item_validation.py", line
29, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual(
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
```

不能完全通过，但取得了一些进展。从 `line 29` 那行能看出，功能测试的第一部分通过了，现在要处理第二部分，即第二次提交空待办事项也要显示错误消息。

不过我们编写了一些可用的代码，那就做个提交吧：

```
$ git commit -am "Adjust new list view to do model validation"
```

13.3 Django模式：在渲染表单的视图中处理POST请求

这一次要使用一种稍微不同的处理方式，这种方式是 Django 中十分常用的模式：在渲染表单的视图中处理该视图接收到的 POST 请求。这么做虽然不太符合 REST 架构的 URL 规

则，却有个很大的好处：同一个 URL 既可以显示表单，又可以显示处理用户输入过程中遇到的错误。

现在的状况是，显示清单用一个视图和 URL，处理新建清单中的待办事项用另一个视图和 URL。要把这两种操作合并到一个视图和 URL 中。所以，在 list.html 中，表单的提交目标地址要改一下：

lists/templates/list.html (ch11l030)

```
{% block form_action %}/lists/{{ list.id }}/{% endblock %}
```

不小心又硬编码了一个 URL，在便签上记下这个地方。回想一下，home.html 中也有一个：



修改之后功能测试随即失败，因为 `view_list` 视图还不知道如何处理 POST 请求：

```
$ python manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

> 本节要进行一次应用层的重构。在应用层中重构时，要先修改或增加单元测试，然后再调整代码。使用功能测试检查重构是否完成，以及一切能否像重构前一样正常运行。如果你想完全理解这个过程，请再看一下第 4 章末尾的图表。

13.3.1 重构：把new_item实现的功能移到view_list中

`NewItemTest` 类中的测试用于检查把 POST 请求中的数据保存到现有的清单中，把这些测试全部移到 `ListViewTest` 类中，还要把原来的请求目标地址 .../add_item 改成显示清单的 URL：

lists/tests/test_views.py (ch11l031)

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        [...]

    def test_passes_correct_list_to_template(self):
        [...]

    def test_displays_only_items_for_that_list(self):
        [...]

    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f'/lists/{correct_list.id}/',
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new item for an existing list')
        self.assertEqual(new_item.list, correct_list)


    def test_POST_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            f'/lists/{correct_list.id}/',
            data={'item_text': 'A new item for an existing list'}
        )
        self.assertRedirects(response, f'/lists/{correct_list.id}/')
```

注意，整个 `NewItemTest` 类都没有了。而且我还修改了重定向测试方法的名字，明确表明只适用于 POST 请求。

改动之后测试的结果为：

```
FAIL: test_POST_redirects_to_list_view (lists.tests.test_views.ListViewTest)
AssertionError: 200 != 302 : Response didn't redirect as expected: Response
code was 200 (expected 302)
[...]
FAIL: test_can_save_a_POST_request_to_an_existing_list
(lists.tests.test_views.ListViewTest)
AssertionError: 0 != 1
```

然后修改 `view_list` 函数，处理两种请求类型：

lists/views.py (ch11l032-1)

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
```

```
        Item.objects.create(text=request.POST['item_text'], list=list_)
        return redirect(f'/lists/{list_.id}/')
    return render(request, 'list.html', {'list': list_})
```

修改之后测试通过了：

```
Ran 12 tests in 0.047s

OK
```

现在可以删除 `add_item` 视图，因为不再需要了。但出乎意料，一些测试失败了：

```
[...]
AttributeError: module 'lists.views' has no attribute 'add_item'
```

失败的原因是，虽然删除了视图，但在 urls.py 中仍然引用这个视图。把引用也删除：

lists/urls.py (ch11l033)

```
urlpatterns = [
    url(r'^new$', views.new_list, name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
]
```

这样单元测试就能通过了。运行所有功能测试看看结果如何：

```
$ python manage.py test
[...]
ERROR: test_cannot_add_empty_list_items
[...]

Ran 16 tests in 15.276s
FAILED (errors=1)
```

依然是那个新功能测试中的失败。至此，重构 `add_item` 功能的任务完成了。此时应该提交代码：

```
$ git commit -am "Refactor list view to handle new item POSTs"
```

> 我是不是破坏了“有测试失败时不重构”这个规则？本节可以这么做，因为若想使用新功能必须重构。如果有**单元**测试失败，决不能重构。不过在本书中，虽然当前这个用户故事的功能测试失败了，但仍然可以重构。[1]

13.3.2　在view_list视图中执行模型验证

把待办事项添加到现有清单时，我们希望保存数据时仍能遵守制定好的模型验证规则。为此要编写一个新单元测试，和首页的单元测试差不多，但有几处不同：

注 1：如果你更想看到一个干净的测试结果，那么可以为这个功能测试方法加上 `@skip` 装饰器，或者在测试方法中尽早返回。但是，别忘了你这么做过。

lists/tests/test_views.py (ch11l034)

```
class ListViewTest(TestCase):
    [...]

    def test_validation_errors_end_up_on_lists_page(self):
        list_ = List.objects.create()
        response = self.client.post(
            f'/lists/{list_.id}/',
            data={'item_text': ''}
        )
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html')
        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

这个测试应该失败，因为视图现在还没做任何验证，只是重定向所有 POST 请求：

```
    self.assertEqual(response.status_code, 200)
AssertionError: 302 != 200
```

在视图中执行验证的方法如下：

lists/views.py (ch11l035)

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    error = None

    if request.method == 'POST':
        try:
            item = Item(text=request.POST['item_text'], list=list_)
            item.full_clean()
            item.save()
            return redirect(f'/lists/{list_.id}/')
        except ValidationError:
            error = "You can't have an empty list item"

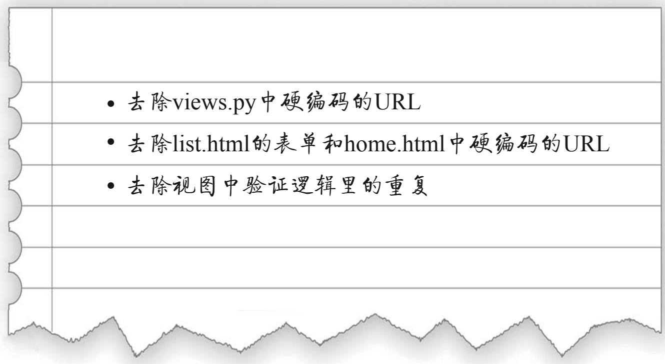
    return render(request, 'list.html', {'list': list_, 'error': error})
```

对这段代码不是特别满意，是吧？确实有一些重复的代码，views.py 中出现了两次 `try/except` 语句，一般来说不太好看。

```
Ran 13 tests in 0.047s

OK
```

稍等一会儿再重构，因为我们知道验证待办事项重复的代码有点不同。先把这件事记在便签上：



> 制定“事不过三，三则重构”这个规则的原因之一是，只有遇到三次且每次都稍有不同时，才能更好地提炼出通用功能。如果过早重构，得到的代码可能并不适用于第三次。

至少功能测试又可以通过了：

```
$ python manage.py test functional_tests
[...]
OK
```

又回到了可正常运行的状态，因此可以看一下便签上的记录了。现在是提交的好时机。或许还可以喝杯茶休息一下。

```
$ git commit -am "enforce model validation in list view"
```

13.4 重构：去除硬编码的URL

还记得 urls.py 中 `name=` 参数的写法吗？我们是直接从 Django 生成的默认 URL 映射中复制过来，然后又给它们起了有意义的名字。现在要查明这些名字有什么用。

lists/urls.py

```
    url(r'^new$', views.new_list, name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
```

13.4.1 模板标签{% url %}

可以把 home.html 中硬编码的 URL 换成一个 Django 模板标签，再引用 URL 的“名字”：

lists/templates/home.html (ch11l036-1)

```
{% block form_action %}{% url 'new_list' %}{% endblock %}
```

然后确认改动之后不会导致单元测试失败：

```
$ python manage.py test lists
OK
```

继续修改其他模板。传入了一个参数，所以这一个更有趣：

lists/templates/list.html (ch11l036-2)

```
{% block form_action %}{% url 'view_list' list.id %}{% endblock %}
```

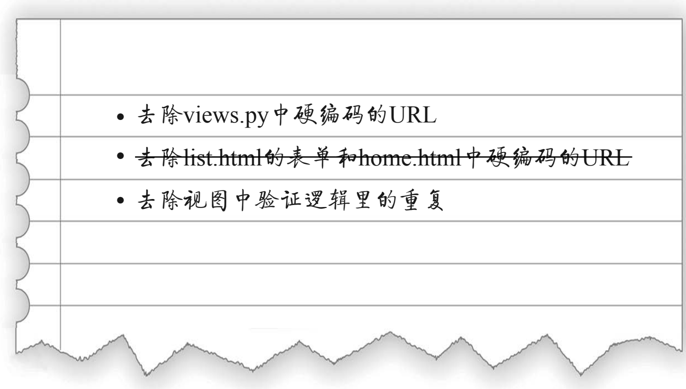
详情请阅读 Django 文档中对 URL 反向解析的介绍。

再次运行测试，确保都能通过：

```
$ python manage.py test lists
OK
$ python manage.py test functional_tests
OK
```

太棒了，做次提交：

```
$ git commit -am "Refactor hard-coded URLs out of templates"
```



13.4.2 重定向时使用get_absolute_url

下面来处理 views.py。在这个文件中去除硬编码的 URL，可以使用和模板一样的方法——写入 URL 的名字和一个位置参数：

lists/views.py (ch11l036-3)

```
def new_list(request):
    [...]
    return redirect('view_list', list_.id)
```

修改之后单元测试和功能测试仍能通过，但是 `redirect` 函数的作用远比这强大。在 Django 中，每个模型对象都对应一个特定的 URL，因此可以定义一个特殊的函数，命名为 `get_absolute_url`，其作用是获取显示单个模型对象的页面 URL。这个函数在这里很有用，在 Django 管理后台（本书不会介绍管理后台，但稍后你可以自己学习）也很有用：

在后台查看一个对象时可以直接跳到前台显示该对象的页面。如果有必要，我总是建议在模型中定义 `get_absolute_url` 函数，这花不了多少时间。

先在 test_models.py 中编写一个超级简单的单元测试：

lists/tests/test_models.py (ch11l036-4)

```
    def test_get_absolute_url(self):
        list_ = List.objects.create()
        self.assertEqual(list_.get_absolute_url(), f'/lists/{list_.id}/')
```

得到的测试结果是：

```
AttributeError: 'List' object has no attribute 'get_absolute_url'
```

实现这个函数时要使用 Django 中的 `reverse` 函数。`reverse` 函数的功能和 Django 对 urls.py 所做的操作相反。

lists/models.py (ch11l036-5)

```
from django.core.urlresolvers import reverse


class List(models.Model):

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])
```

现在可以在视图中使用 `get_absolute_url` 函数了——只需把重定向的目标对象传给 `redirect` 函数即可，`redirect` 函数会自动调用 `get_absolute_url` 函数。

lists/views.py (ch11l036-6)

```
def new_list(request):
    [...]
    return redirect(list_)
```

更多信息参见 Django 文档。快速确认一下单元测试是否仍能通过：

```
OK
```

然后使用同样的方法修改 `view_list` 视图：

lists/views.py (ch11l036-7)

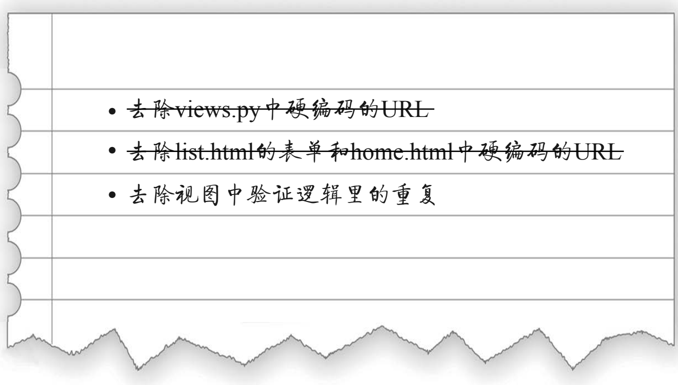
```
def view_list(request, list_id):
    [...]

            item.save()
            return redirect(list_)
        except ValidationError:
            error = "You can't have an empty list item"
```

分别运行全部单元测试和功能测试，确保一切仍能正常运行：

```
$ python3 manage.py test lists
OK
$ python3 manage.py test functional_tests
OK
```

把已解决问题从便签上划掉：



提交一次：

```
$ git commit -am "Use get_absolute_url on List model to DRY urls in views"
```

这一阶段结束！我们添加了模型层验证，而且在此过程中借机重构了几个地方。

最后一个待办事项是下一章的话题。

> **关于数据库层验证**
>
> 我喜欢尽量把验证逻辑放在低层。
>
> - 数据库层验证是数据完整性的最终保障
>
>   不管数据库层之上的各层代码有多么复杂，在最低层验证能保证数据是有效的，而且是一致的。
>
> - 但是数据库层验证有失灵活性
>
>   优点往往都伴随着缺点。添加数据库层验证之后，就无法得到不一致的数据了，即便想暂时这么做也不可能。但我们有时就需要存储暂时破坏这些规则的数据（例如在很多阶段可能想从外部源导入数据），毕竟有数据总比没数据强。
>
> - 对用户不太友好
>
>   尝试存储无效数据会导致数据库返回不友善的 IntegrityError，这可能会让用户看到令人困惑的 500 错误页面。后面的章节会讲到，表单层验证考虑到了用户，不会直接报错，而是显示友好的错误消息。

第 14 章

简单的表单

前一章结尾提到，视图中处理验证的代码有太多重复。Django 鼓励使用表单类验证用户的输入，以及选择显示错误消息。本章介绍如何使用这种功能。

除此之外本章还会花点时间整理单元测试，确保一个单元测试一次只测试一件事。

14.1 把验证逻辑移到表单中

> 在 Django 中，视图很复杂就说明有代码异味。你要想，能否把逻辑移到表单或模型类的方法中，或者把业务逻辑移到 Django 之外的模型中？

Django 中的表单功能很多很强大。

- 可以处理用户输入，并验证输入值是否有错误。
- 可以在模板中使用，用来渲染 HTML `input` 元素和错误消息。
- 稍后会见识到，某些表单甚至还可以把数据存入数据库。

没必要在每个表单中都使用这三种功能。你可以自己编写表单的 HTML，或者自己处理数据存储，但表单是放置验证逻辑的绝佳位置。

14.1.1　使用单元测试探索表单API

我们要在一个单元测试中实验表单的用法。我的计划是逐步迭代，最终得到一个完整的解决方案。希望在这个过程中能由浅入深地介绍表单，即便你以前从未用过也能理解。

首先，新建一个文件，用于编写表单的单元测试。先编写一个测试方法，检查表单的 HTML：

lists/tests/test_forms.py

```
from django.test import TestCase

from lists.forms import ItemForm


class ItemFormTest(TestCase):

    def test_form_renders_item_text_input(self):
        form = ItemForm()
        self.fail(form.as_p())
```

`form.as_p()` 的作用是把表单渲染成 HTML。这个单元测试使用 `self.fail` 探索性编程。在 `manage.py shell` 会话中探索编程也很容易，不过每次修改代码之后都要重新加载。

下面编写一个极简的表单，继承自基类 `Form`，只有一个字段 `item_text`：

lists/forms.py

```
from django import forms

class ItemForm(forms.Form):
    item_text = forms.CharField()
```

运行测试后会看到一个失败消息，告诉我们自动生成的表单 HTML 是什么样：

```
    self.fail(form.as_p())
AssertionError: <p><label for="id_item_text">Item text:</label> <input
type="text" name="item_text" required id="id_item_text" /></p>
```

自动生成的 HTML 已经和 base.html 中的表单 HTML 很接近了，只不过没有 `placeholder` 属性和 Bootstrap 的 CSS 类。再编写一个单元测试方法，检查 `placeholder` 属性和 CSS 类：

lists/tests/test_forms.py

```
class ItemFormTest(TestCase):

    def test_form_item_input_has_placeholder_and_css_classes(self):
        form = ItemForm()
        self.assertIn('placeholder="Enter a to-do item"', form.as_p())
        self.assertIn('class="form-control input-lg"', form.as_p())
```

这个测试会失败，表明我们需要真正地编写一些代码了。应该怎么定制表单字段的内容呢？答案是使用 `widget` 参数。加入 `placeholder` 属性的方法如下：

lists/forms.py

```
class ItemForm(forms.Form):
    item_text = forms.CharField(
        widget=forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
        }),
    )
```

修改之后测试的结果为：

```
AssertionError: 'class="form-control input-lg"' not found in '<p><label
for="id_item_text">Item text:</label> <input type="text" name="item_text"
placeholder="Enter a to-do item" required id="id_item_text" /></p>'
```

继续修改：

lists/forms.py

```
    widget=forms.fields.TextInput(attrs={
        'placeholder': 'Enter a to-do item',
        'class': 'form-control input-lg',
    }),
```

> 如果表单中的内容很多或者很复杂，使用 `widget` 参数定制很麻烦，此时可以借助 django-crispy-forms 和 django-floppyforms。

> **开发驱动测试：使用单元测试探索性编程**
>
> 上述过程是不是有点像开发驱动测试？偶尔这么做其实没问题。
>
> 探索新 API 时，完全可以先抛开规则的束缚，然后再回到严格的 TDD 流程中。你可以使用交互式终端，或者编写一些探索性代码（不过你要答应测试山羊，稍后会删掉这些代码，然后使用合理的方式重写）。
>
> 其实，现在我们只是使用单元测试试验表单 API，这是学习如何使用 API 的好方法。

14.1.2　换用Django中的`ModelForm`类

接下来呢？我们希望表单重用已经在模型中定义好的验证规则。Django 提供了一个特殊的类，用来自动生成模型的表单，这个类是 `ModelForm`。从下面的代码能看出，我们要使用一个特殊的属性 `Meta` 配置表单：

lists/forms.py

```
from django import forms

from lists.models import Item
```

```
class ItemForm(forms.models.ModelForm):

    class Meta:
        model = Item
        fields = ('text',)
```

我们在 `Meta` 中指定这个表单用于哪个模型，以及要使用哪些字段。

`ModelForms` 很智能，能完成各种操作，例如为不同类型的字段生成合适的 `input` 类型，以及应用默认的验证。详情参见文档（https://docs.djangoproject.com/en/1.11/topics/forms/modelforms/）。

现在表单的 HTML 不一样了：

```
AssertionError: 'placeholder="Enter a to-do item"' not found in '<p><label
for="id_text">Text:</label> <textarea name="text" cols="40" rows="10" required
id="id_text">\n</textarea></p>'
```

`placeholder` 属性和 CSS 类都不见了，而且 `name="item_text"` 变成了 `name="text"`。这些变化能接受，但普通的输入框变成了 `textarea`，这可不是应用 UI 想要的效果。幸好，和普通的表单类似，`ModelForm` 的字段也能使用 `widget` 参数定制：

lists/forms.py

```
class ItemForm(forms.models.ModelForm):

    class Meta:
        model = Item
        fields = ('text',)
        widgets = {
            'text': forms.fields.TextInput(attrs={
                'placeholder': 'Enter a to-do item',
                'class': 'form-control input-lg',
            }),
        }
```

定制后测试通过了。

14.1.3 测试和定制表单验证

现在我们看一下 `ModelForm` 是否应用了模型中定义的验证规则。我们还会学习如何把数据传入表单，就像用户输入的一样：

lists/tests/test_forms.py (ch11l008)

```
    def test_form_validation_for_blank_items(self):
        form = ItemForm(data={'text': ''})
        form.save()
```

测试的结果为：

```
ValueError: The Item could not be created because the data didn't validate.
```

很好，如果提交空待办事项，表单不会保存数据。

现在看一下表单能否显示指定的错误消息。在尝试保存数据**之前**检查验证是否通过的 API 是 `is_valid` 函数：

lists/tests/test_forms.py (ch11l009)

```
def test_form_validation_for_blank_items(self):
    form = ItemForm(data={'text': ''})
    self.assertFalse(form.is_valid())
    self.assertEqual(
        form.errors['text'],
        ["You can't have an empty list item"]
    )
```

调用 `form.is_valid()` 得到的返回值是 `True` 或 `False`，不过还有个附带效果，即验证输入的数据，生成 `errors` 属性。`errors` 是个字典，把字段的名字映射到该字段的错误列表上（一个字段可以有多个错误）。

测试的结果为：

```
AssertionError: ['This field is required.'] != ["You can't have an empty list
item"]
```

Django 已经为显示给用户查看的错误消息提供了默认值。急着开发 Web 应用的话，可以直接使用默认值。不过我们比较在意，想让错误消息特殊一些。定制错误消息可以修改 `Meta` 的另一个变量，`error_messages`：

lists/forms.py (ch11l010)

```
    class Meta:
        model = Item
        fields = ('text',)
        widgets = {
            'text': forms.fields.TextInput(attrs={
                'placeholder': 'Enter a to-do item',
                'class': 'form-control input-lg',
            }),
        }
        error_messages = {
            'text': {'required': "You can't have an empty list item"}
        }
```

然后测试即可通过：

```
OK
```

知道如何避免让这些错误消息搅乱代码吗？使用常量：

lists/forms.py (ch11l011)

```
EMPTY_ITEM_ERROR = "You can't have an empty list item"
[...]

        error_messages = {
            'text': {'required': EMPTY_ITEM_ERROR}
        }
```

再次运行测试，确认能通过。好的，然后修改测试：

lists/tests/test_forms.py (ch11l012)

```
from lists.forms import EMPTY_ITEM_ERROR, ItemForm
[...]

    def test_form_validation_for_blank_items(self):
        form = ItemForm(data={'text': ''})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])
```

修改之后测试仍能通过：

```
OK
```

很好。提交：

```
$ git status # 会看到lists/forms.py和tests/test_forms.py
$ git add lists
$ git commit -m "new form for list items"
```

14.2 在视图中使用这个表单

一开始我想继续编写这个表单，除了空值验证之外再捕获唯一性验证错误。不过，精益理论中的“尽早部署”有个推论，即“尽早合并代码”。也就是说，编写表单可能要花很多时间，不断添加各种功能——我知道这一点是因为我在撰写本章草稿时就是这么做的，做了各种工作，得到一个功能完善的表单类，但发布应用后才发现大多数功能实际并不需要。

因此，要尽早试用新编写的代码。这么做能避免编写用不到的代码，还能尽早在现实的环境中检验代码。

我们编写了一个表单类，它可以渲染一些 HTML，而且至少能验证一种错误——下面就来使用这个表单吧！既然可以在 base.html 模板中使用这个表单，那么在所有视图中都可以使用。

14.2.1 在处理GET请求的视图中使用这个表单

先修改首页视图的单元测试。我们要编写一个新测试，检查使用的表单类型是否正确：

lists/tests/test_views.py (ch11l013)

```
from lists.forms import ItemForm

class HomePageTest(TestCase):
```

```
    def test_uses_home_template(self):
        [...]

    def test_home_page_uses_item_form(self):
        response = self.client.get('/')
        self.assertIsInstance(response.context['form'], ItemForm)  ❶
```

❶ assertIsInstance 检查表单是否属于正确的类。

测试的结果为：

```
KeyError: 'form'
```

因此，要在首页视图中使用这个表单：

lists/views.py (ch11l014)

```
[...]
from lists.forms import ItemForm
from lists.models import Item, List

def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

好了，下面尝试在模板中使用这个表单——把原来的 <input ..> 替换成 {{ form.text }}：

lists/templates/base.html (ch11l015)

```
  <form method="POST" action="{% block form_action %}{% endblock %}">
    {{ form.text }}
    {% csrf_token %}
    {% if error %}
      <div class="form-group has-error">
```

{{ form.text }} 只会渲染这个表单中的 text 字段，生成 HTML input 元素。

14.2.2 大量查找和替换

前文我们修改了表单，id 和 name 属性的值变了。运行功能测试时你会看到，首次尝试查找输入框时测试失败了：

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]
```

我们得修正这个问题，为此要进行大量查找和替换。在此之前先提交，把重命名和逻辑变动区分开：

```
$ git diff # 审查base.html、views.py及其测试中的改动
$ git commit -am "use new form in home_page, simplify tests. NB breaks stuff"
```

下面来修正功能测试。通过 grep 命令，我们得知有很多地方都使用了 id_new_item：

```
$ grep id_new_item functional_tests/test*
functional_tests/test_layout_and_styling.py:        inputbox =
```

```
self.browser.find_element_by_id('id_new_item')
functional_tests/test_layout_and_styling.py:        inputbox =
self.browser.find_element_by_id('id_new_item')
functional_tests/test_list_item_validation.py:
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
[...]
```

这表明我们要重构。在 base.py 中定义一个新辅助方法：

functional_tests/base.py (ch11l018)

```
class FunctionalTest(StaticLiveServerTestCase):
    [...]
    def get_item_input_box(self):
        return self.browser.find_element_by_id('id_text')
```

然后所有需要替换的地方都使用这个辅助方法——test_simple_list_creation.py 修改四处，test_layout_and_styling.py 修改两处，test_list_item_validation.py 修改四处。例如：

functional_tests/test_simple_list_creation.py

```
        # 应用邀请她输入一个待办事项
        inputbox = self.get_item_input_box()
```

以及：

functional_tests/test_list_item_validation.py

```
        # 输入框中没输入内容，她就按下了回车键
        self.browser.get(self.live_server_url)
        self.get_item_input_box().send_keys(Keys.ENTER)
```

我不会列出每一处，相信你自己能搞定！你可以再执行一遍 `grep`，看是不是全都改了。

第一步完成了，接下来还要修改应用代码。我们要找到所有旧的 `id`（`id_new_item`）和 `name`（`item_text`），分别替换成 `id_text` 和 `text`：

```
$ grep -r id_new_item lists/
lists/static/base.css:#id_new_item {
```

只要改动一处。使用类似的方法查看 `name` 出现的位置：

```
$ grep -Ir item_text lists
[...]
lists/views.py:    item = Item(text=request.POST['item_text'], list=list_)
lists/views.py:            item = Item(text=request.POST['item_text'],
list=list_)
lists/tests/test_views.py:        self.client.post('/lists/new',
data={'item_text': 'A new list item'})
lists/tests/test_views.py:        response = self.client.post('/lists/new',
data={'item_text': 'A new list item'})
[...]
lists/tests/test_views.py:            data={'item_text': ''}
[...]
```

改完之后再运行单元测试，确保一切仍能正常运行：

```
$ python manage.py test lists
[...]
.................
 ---------------------------------------------------------------------
Ran 17 tests in 0.126s

OK
```

然后还要运行功能测试：

```
$ python manage.py test functional_tests
[...]
  File "/.../superlists/functional_tests/test_simple_list_creation.py", line
37, in test_can_start_a_list_for_one_user
    return self.browser.find_element_by_id('id_text')
  File "/.../superlists/functional_tests/base.py", line 51, in
get_item_input_box
    return self.browser.find_element_by_id('id_text')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_text"]
[...]
FAILED (errors=3)
```

不能全部通过。确认一下发生错误的位置——查看其中一个失败所在的行号，你会发现，每次提交第一个待办事项后，清单页面都不会显示输入框。

查看 views.py 和 `new_list` 视图后我们找到了原因——如果检测到有验证错误，根本就不会把表单传入 home.html 模板：

lists/views.py

```
except ValidationError:
    list_.delete()
    error = "You can't have an empty list item"
    return render(request, 'home.html', {"error": error})
```

我们也想在这个视图中使用 `ItemForm` 表单。继续修改之前，先提交：

```
$ git status
$ git commit -am "rename all item input ids and names. still broken"
```

14.3 在处理POST请求的视图中使用这个表单

现在要调整 `new_list` 视图的单元测试，更确切地说，要修改针对验证的那个测试方法。先看一下这个测试方法：

lists/tests/test_views.py

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
```

```
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

14.3.1　修改new_list视图的单元测试

首先，这个测试方法测试的内容太多了，所以借此机会可以清理一下。我们应该把这个测试方法分成两个不同的断言。

- 如果有验证错误，应该渲染首页模板，并且返回 200 响应。
- 如果有验证错误，响应中应该包含错误消息。

此外，还可以添加一个新断言。

- 如果有验证错误，应该把表单对象传入模板。

不用硬编码错误消息字符串，而要使用一个常量：

lists/tests/test_views.py (ch11l023)

```
from lists.forms import ItemForm, EMPTY_ITEM_ERROR
[...]

class NewListTest(TestCase):
    [...]

    def test_for_invalid_input_renders_home_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')


    def test_validation_errors_are_shown_on_home_page(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))


    def test_for_invalid_input_passes_form_to_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertIsInstance(response.context['form'], ItemForm)
```

现在好多了，每个测试方法只测试一件事。如果幸运的话，只有一个测试会失败，而且会告诉我们接下来做什么：

```
$ python manage.py test lists
[...]
======================================================================
ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_views.py", line 49, in
test_for_invalid_input_passes_form_to_template
```

```
    self.assertIsInstance(response.context['form'], ItemForm)
[...]
KeyError: 'form'

 ---------------------------------------------------------------------
Ran 19 tests in 0.041s

FAILED (errors=1)
```

14.3.2　在视图中使用这个表单

在视图中使用这个表单的方法如下：

lists/views.py

```
def new_list(request):
    form = ItemForm(data=request.POST)   ❶
    if form.is_valid():   ❷
        list_ = List.objects.create()
        Item.objects.create(text=request.POST['text'], list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})   ❸
```

❶ 把 `request.POST` 中的数据传给表单的构造方法。

❷ 使用 `form.is_valid()` 判断提交是否成功。

❸ 如果提交失败，把表单对象传入模板，而不显示一个硬编码的错误消息字符串。

视图现在看起来更完美了。而且除了一个测试之外，其他测试都能通过：

```
    self.assertContains(response, escape(EMPTY_ITEM_ERROR))
[...]
AssertionError: False is not true : Couldn't find 'You can&#39;t have an empty
list item' in response
```

14.3.3　使用这个表单在模板中显示错误消息

测试失败的原因是模板还没使用这个表单显示错误消息：

lists/templates/base.html (ch11l026)

```
  <form method="POST" action="{% block form_action %}{% endblock %}">
    {{ form.text }}
    {% csrf_token %}
    {% if form.errors %}   ❶
      <div class="form-group has-error">
        <div class="help-block">{{ form.text.errors }}</div>   ❷
      </div>
    {% endif %}
  </form>
```

❶ `form.errors` 是一个列表，包含这个表单中的所有错误。

❷ `form.text.errors` 也是一个列表，但只包含 `text` 字段的错误。

这样修改之后对测试有什么作用呢?

```
FAIL: test_validation_errors_end_up_on_lists_page
(lists.tests.test_views.ListViewTest)
[...]
AssertionError: False is not true : Couldn't find 'You can&#39;t have an empty
list item' in response
```

得到了一个意料之外的失败，这次失败发生在针对最后一个试图 `view_list` 的测试中。因为我们修改了错误在**所有**模板中显示的方式，不再显示手动传入模板的错误。

因此，还要修改 `view_list` 视图才能重新回到可运行状态。

14.4 在其他视图中使用这个表单

`view_list` 视图既可以处理 GET 请求也可以处理 POST 请求。先测试 GET 请求，为此，可以编写一个新测试方法：

lists/tests/test_views.py

```
class ListViewTest(TestCase):
    [...]

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertIsInstance(response.context['form'], ItemForm)
        self.assertContains(response, 'name="text"')
```

测试的结果为：

```
KeyError: 'form'
```

解决这个问题最简单的方法如下：

lists/views.py (ch11l028)

```
def view_list(request, list_id):
    [...]
    form = ItemForm()
    return render(request, 'list.html', {
        'list': list_, "form": form, "error": error
    })
```

14.4.1 定义辅助方法，简化测试

接下来要在另一个视图中使用这个表单的错误消息，把当前针对表单提交失败的测试（`test_validation_errors_end_up_on_lists_page`）分成多个测试方法：

lists/tests/test_views.py (ch11l030)

```
class ListViewTest(TestCase):
    [...]
```

```
    def post_invalid_input(self):
        list_ = List.objects.create()
        return self.client.post(
            f'/lists/{list_.id}/',
            data={'text': ''}
        )

    def test_for_invalid_input_nothing_saved_to_db(self):
        self.post_invalid_input()
        self.assertEqual(Item.objects.count(), 0)

    def test_for_invalid_input_renders_list_template(self):
        response = self.post_invalid_input()
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html')

    def test_for_invalid_input_passes_form_to_template(self):
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ItemForm)

    def test_for_invalid_input_shows_error_on_page(self):
        response = self.post_invalid_input()
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))
```

我们定义了一个辅助方法 `post_invalid_input`，这样就不用在分拆的四个测试中重复编写代码了。

这种做法我们见过几次了。把视图测试写在一个测试方法中，编写一连串的断言检测视图应该做这个、这个和这个，然后应该返回那个——我们经常觉得这么做更合理，但把单个测试方法分解为多个方法也绝对有好处。从前面几章我们已经得知，如果以后修改代码时不小心引入了一个问题，分拆的测试能帮助定位真正的问题所在。辅助方法则是降低心理障碍的方式之一。

例如，现在测试结果只有一个失败，而且我们知道是由哪个测试方法导致的：

```
FAIL: test_for_invalid_input_shows_error_on_page
(lists.tests.test_views.ListViewTest)
AssertionError: False is not true : Couldn't find 'You can&#39;t have an empty
list item' in response
```

现在，试试能否使用 `ItemForm` 表单重写视图。第一次尝试：

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
        if form.is_valid():
            Item.objects.create(text=request.POST['text'], list=list_)
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

重写后，单元测试通过了：

```
Ran 23 tests in 0.086s

OK
```

再看功能测试的结果如何：

```
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
File "/.../superlists/functional_tests/test_list_item_validation.py", line
15, in test_cannot_add_empty_list_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
```

失败。

14.4.2 意想不到的好处：HTML5自带的客户端验证

这是怎么回事呢？我们在错误所处位置之前加上 `time.sleep`，看看会发生什么（如果愿意，也可以执行 `manage.py runserver` 命令，自己动手访问网站，如图 14-1 所示）。

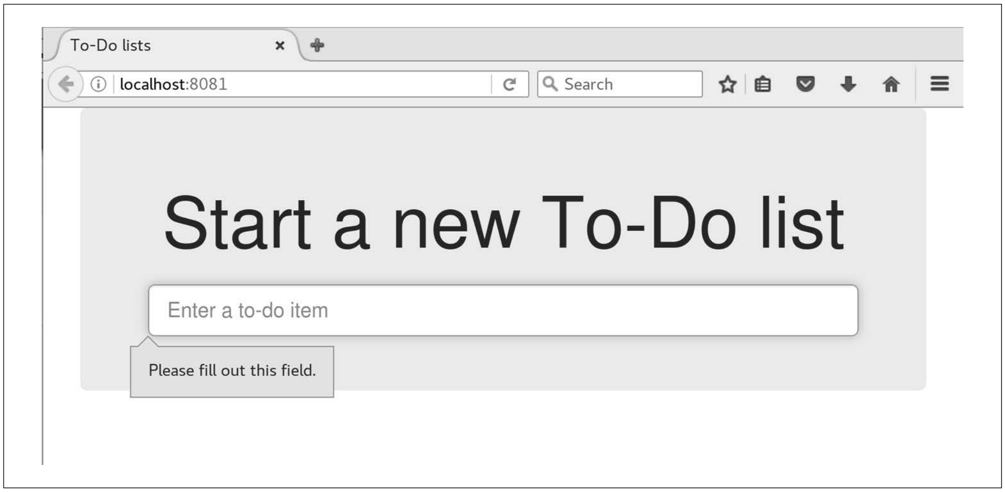


图 14-1：HTML5 验证失败

看起来输入框为空时，浏览器禁止用户提交表单。

这是因为 Django 为那个 HTML 输入框添加了 `required` 属性[1]。（不相信？再看一下前面的 `as_p()` 输出。）这是 HTML5 的新特性，浏览器会在客户端做些验证，输入无效时禁止用户提交表单。

注 1：这是 Django 1.11 的新特性。

下面据此修改功能测试：

functional_tests/test_list_item_validation.py (ch11l032)

```
    def test_cannot_add_empty_list_items(self):
        # 伊迪丝访问首页，不小心提交了一个空待办事项
        # 输入框中没输入内容，她就按下了回车键
        self.browser.get(self.live_server_url)
        self.get_item_input_box().send_keys(Keys.ENTER)

        # 浏览器截获了请求
        # 清单页面不会加载
        self.wait_for(lambda: self.browser.find_elements_by_css_selector(
            '#id_text:invalid'  ❶
        ))

        # 她在待办事项中输入了一些文字
        # 错误消失了
        self.get_item_input_box().send_keys('Buy milk')
        self.wait_for(lambda: self.browser.find_elements_by_css_selector(
            '#id_text:valid'  ❷
        ))

        # 现在能提交了
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table('1: Buy milk')

        # 她有点儿调皮，打算再提交一个空待办事项
        self.get_item_input_box().send_keys(Keys.ENTER)

        # 浏览器这次也不会放行
        self.wait_for_row_in_list_table('1: Buy milk')
        self.wait_for(lambda: self.browser.find_elements_by_css_selector(
            '#id_text:invalid'
        ))

        # 输入一些文字后就能纠正这个错误
        self.get_item_input_box().send_keys('Make tea')
        self.wait_for(lambda: self.browser.find_elements_by_css_selector(
            '#id_text:valid'
        ))
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table('1: Buy milk')
        self.wait_for_row_in_list_table('2: Make tea')
```

❶ 不再检查我们自定义的错误消息，而是通过 CSS 伪选择符 `:invalid` 检查。这个伪选择符是浏览器为输入无效内容的 HTML5 输入框添加的。

❷ 输入有效的内容时，伪选择符逆转。

看到 `self.wait_for` 函数多么有用、多么灵活了吗？

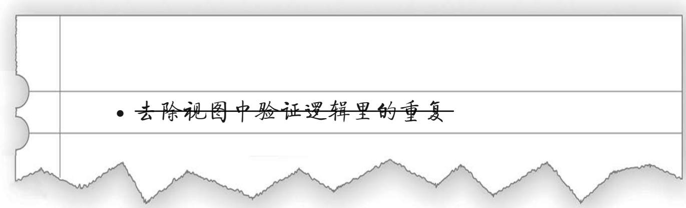
现在的功能测试与刚开始时区别很大，我相信此时此刻你有很多疑问。先别急，我会说明的。先来看看测试是否又能通过了：

```
$ python manage.py test functional_tests
[...]
....
 ---------------------------------------------------------------------
Ran 4 tests in 12.154s

OK
```

14.5　值得鼓励

首先，给自己一个大大的肯定：我们刚刚完成了这个小型应用中的一项重要修改——那个输入框，以及它的 name 和 id 属性，对应用正常运行至关重要。我们修改了七八个文件，完成了一次工作量很大的重构。如果没有测试，做这么复杂的重构我一定会担心，甚至有可能觉得没必要再去改动可以使用的代码。可是，我们有一套完整的测试组件，所以可以深入研究、整理代码，这些操作都很安全，因为我们知道如果有错误，测试能发现。所以我们会不断重构、整理和维护代码，确保整个应用的代码干净整洁，运行起来毫无障碍、准确无误，而且功能完善。



现在是提交的绝佳时刻：

```
$ git diff
$ git commit -am "use form in all views, back to working state"
```

14.6　这难道不是浪费时间吗

如果这样的话，我们自定义的错误消息还有什么用呢？我们在 HTML 模板中费这么大力气渲染表单都是无用功吗？如果在产生错误之前，浏览器就截获了请求，Django 根本无法把错误呈现到用户面前，功能测试也就无从测试。

好吧，你说得对。但是我们的时间并没有浪费，原因有三个。首先，客户端验证不能百分百阻止无效输入。如果你真的在意数据完整性，就必须使用服务器端验证，而这部分逻辑很适合封装在表单中。

其次，不是所有浏览器（咳，Safari）都完全支持 HTML5，有些用户还是能看到我们自定义的消息的。而且，如果我们打算让用户通过 API 访问数据（参见附录 F），验证消息也会回送给用户。

此外，下一章将重用这里的验证、表单代码以及前端 .has-error 类，实现一些 HTML5 没有的高级验证。

话说回来，就算没有这些理由，你也不用为编程时走错路了而责怪自己。没人能预见未来，我们的目标是找出正确的解决方案，不惜“浪费”时间在错误的方案上。

14.7　使用表单自带的save方法

我们还可以进一步简化视图。前面说过，表单可以把数据存入数据库。我们遇到的情况并不能直接保存数据，因为需要知道把待办事项保存到哪个清单中，不过解决起来也不难。

一如既往，先编写测试。为了查明遇到的问题，先看一下如果直接调用 `form.save()` 会发生什么：

lists/tests/test_forms.py (ch11l033)

```
    def test_form_save_handles_saving_to_a_list(self):
        form = ItemForm(data={'text': 'do me'})
        new_item = form.save()
```

Django 报错了，因为待办事项必须隶属于某个清单：

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

这个问题的解决办法是告诉表单的 `save` 方法，应该把待办事项保存到哪个清单中：

lists/tests/test_forms.py

```
from lists.models import Item, List
[...]

    def test_form_save_handles_saving_to_a_list(self):
        list_ = List.objects.create()
        form = ItemForm(data={'text': 'do me'})
        new_item = form.save(for_list=list_)
        self.assertEqual(new_item, Item.objects.first())
        self.assertEqual(new_item.text, 'do me')
        self.assertEqual(new_item.list, list_)
```

然后，要保证待办事项能顺利存入数据库，而且各个属性的值都正确：

```
TypeError: save() got an unexpected keyword argument 'for_list'
```

可以定制 `save` 方法，实现方式如下：

lists/forms.py (ch11l035)

```
    def save(self, for_list):
        self.instance.list = for_list
        return super().save()
```

表单的 `.instance` 属性是将要修改或创建的数据库对象。我也是在撰写本章时才知道这种用法的。此外还有很多方法，例如自己手动创建数据库对象，或者调用 `save()` 方法时指定参数 `commit=False`，但我觉得使用 `.instance` 属性最简洁。下一章还会介绍一种方法，让表单知道它应用于哪个清单。

```
Ran 24 tests in 0.086s
OK
```

最后，要重构视图。先重构 `new_list`：

lists/views.py

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

然后运行测试，确保都能通过：

```
Ran 24 tests in 0.086s
OK
```

接着重构 `view_list`：

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
        if form.is_valid():
            form.save(for_list=list_)
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

修改之后，单元测试仍能全部通过：

```
Ran 24 tests in 0.111s
OK
```

功能测试也能通过：

```
Ran 4 tests in 14.367s
OK
```

太棒了！现在这两个视图更像是“正常的”Django 视图了：从用户的请求中读取数据，结合一些定制的逻辑或 URL 中的信息（`list_id`），然后把数据传入表单进行验证，如果通过验证就保存数据，最后重定向或者渲染模板。

表单和验证在 Django 以及常规的 Web 编程中都很重要，下一章要看一下能否编写稍微复杂的表单。

小贴士

- 简化视图

  如果发现视图很复杂，要编写很多测试，这时候就应该考虑是否能把逻辑移到其他地方。可以移到表单中，就像本章中的做法一样；也可以移到模型类的自定义方法中。如果应用本身就很复杂，可以把核心业务逻辑移到 Django 专属的文件之外，编写单独的类和函数。

- 一个测试只测试一件事

  如果一个测试中不止一个断言，你就要怀疑这么写是否合理。有时断言之间联系紧密，可以放在一起。不过第一次编写测试时往往都会测试很多表现，其实应该把它们分成多个测试。辅助函数有助于简化拆分后的测试。

第 15 章

高级表单

接下来，你将看到表单的一些高级用法。我们已经帮助用户避免输入空待办事项，接下来要避免用户输入重复的待办事项。

本章将进一步介绍 Django 表单验证的细节。如果你已经完全了解如何定制 Django 表单，或者你阅读本书的目的是学习 TDD 而不是 Django，那就可以跳过本章。

如果你还想接着学习 Django，本章有些值得学习的重要知识。如果你想跳过本章也可以，不过一定要快速阅读关于开发者犯错的框注和本章末尾对视图测试的总结。

15.1　针对重复待办事项的功能测试

在 `ItemValidationTest` 类中再添加一个测试方法：

functional_tests/test_list_item_validation.py (ch13l001)

```
    def test_cannot_add_duplicate_items(self):
        # 伊迪丝访问首页，新建一个清单
        self.browser.get(self.live_server_url)
        self.get_item_input_box().send_keys('Buy wellies')
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table('1: Buy wellies')

        # 她不小心输入了一个重复的待办事项
        self.get_item_input_box().send_keys('Buy wellies')
        self.get_item_input_box().send_keys(Keys.ENTER)

        # 她看到一条有帮助的错误消息
        self.wait_for(lambda: self.assertEqual(
            self.browser.find_element_by_css_selector('.has-error').text,
```

```
            "You've already got this in your list"
        ))
```

为什么编写两个测试方法，而不直接在原来的基础上扩展，或者新建一个文件和类？要自己判断该怎么做。这两种方法看起来联系紧密，都和同一个输入字段的验证有关，所以放在同一个文件中没问题。另一方面，这两种方法在逻辑上互相独立，所以将它们设为不同的两种不同的方法是可行的：

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error

Ran 2 tests in 9.613s
```

好的，这两个测试中的第一个现在可以通过。你可能会问："有没有办法只运行那个失败的测试？"确实有：

```
$ python manage.py test functional_tests.\
test_list_item_validation.ItemValidationTest.test_cannot_add_duplicate_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
```

15.1.1　在模型层禁止重复

这是我们真正要做的事情。编写一个新测试，检查同一个清单中有重复的待办事项时是否抛出异常：

lists/tests/test_models.py (ch09l028)

```
def test_duplicate_items_are_invalid(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='bla')
    with self.assertRaises(ValidationError):
        item = Item(list=list_, text='bla')
        item.full_clean()
```

此外，还要再添加一个测试，确保完整性约束不要做过头了：

lists/tests/test_models.py (ch09l029)

```
def test_CAN_save_same_item_to_different_lists(self):
    list1 = List.objects.create()
    list2 = List.objects.create()
    Item.objects.create(list=list1, text='bla')
    item = Item(list=list2, text='bla')
    item.full_clean() # 不该抛出异常
```

我总喜欢在检查某项操作**不**该抛出异常的测试中加入一些注释，要不然很难看出在测试什么。

```
AssertionError: ValidationError not raised
```

如果想故意犯错，可以这么做：

lists/models.py (ch09l030)

```
class Item(models.Model):
    text = models.TextField(default='', unique=True)
    list = models.ForeignKey(List, default=None)
```

这么做可以确认第二个测试确实能检测到这个问题：

```
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_models.py", line 62, in
test_CAN_save_same_item_to_different_lists
    item.full_clean() # 不该抛出异常
    [...]
django.core.exceptions.ValidationError: {'text': ['Item with this Text already
exists.']}
```

何时测试开发者犯下的错误

测试时要判断何时应该编写测试确认我们没有犯错。一般而言，做决定时要谨慎。

这里，编写测试确认无法把重复的待办事项存入同一个清单。目前，让这个测试通过最简单的方法（即编写的代码量最少）是，让表单无法保存**任何**重复的待办事项。此时就要编写另一个测试，因为我们编写的代码可能有错。

但是，不可能编写测试检查所有可能出错的方式。如果有一个函数计算两数之和，可以编写一些测试：

```
assert adder(1, 1) == 2
assert adder(2, 1) == 3
```

但不应该认为实现这个函数时故意编写了有违常理的代码：

```
def adder(a, b):
    # 不可能这么写！
    if a == 3:
        return 666
    else:
        return a + b
```

判断时你要相信自己不会**故意**犯错，只会**不小心**犯错。

模型和 `ModelForm` 一样，也能使用 `class Meta`。在 `Meta` 类中可以实现一个约束，要求清单中的待办事项必须是唯一的。也就是说，`text` 和 `list` 的组合必须是唯一的：

lists/models.py (ch09l031)

```
class Item(models.Model):
    text = models.TextField(default='')
    list = models.ForeignKey(List, default=None)

    class Meta:
        unique_together = ('list', 'text')
```

此时，你可能想快速浏览一遍 Django 文档中对模型属性 `Meta` 的说明。

15.1.2　题外话：查询集合排序和字符串表示形式

运行测试，会看到一个意料之外的失败：

```
======================================================================
FAIL: test_saving_and_retrieving_items
(lists.tests.test_models.ListAndItemModelsTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_models.py", line 31, in
test_saving_and_retrieving_items
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
AssertionError: 'Item the second' != 'The first (ever) list item'
- Item the second
[...]
```

根据所用系统和 SQLite 版本的不同，你可能看不到这个错误。如果没看到就直接阅读下一节，代码和测试本身也很有趣。

失败消息有点儿晦涩。输出一些信息，以便调试：

lists/tests/test_models.py

```
        first_saved_item = saved_items[0]
        print(first_saved_item.text)
        second_saved_item = saved_items[1]
        print(second_saved_item.text)
        self.assertEqual(first_saved_item.text, 'The first (ever) list item')
```

然后，看到的测试结果如下：

```
.....Item the second
The first (ever) list item
F.....
```

看样子唯一性约束干扰了查询（例如 `Item.objects.all()`）的默认排序。虽然现在仍有测试失败，但最好添加一个新测试明确测试排序：

lists/tests/test_models.py (ch09l032)

```
    def test_list_ordering(self):
        list1 = List.objects.create()
        item1 = Item.objects.create(list=list1, text='i1')
        item2 = Item.objects.create(list=list1, text='item 2')
        item3 = Item.objects.create(list=list1, text='3')
        self.assertEqual(
            Item.objects.all(),
            [item1, item2, item3]
        )
```

测试的结果多了一个失败，而且也不易读：

```
AssertionError: <QuerySet [<Item: Item object>, <Item: Item object>, <Item:
Item object>]> != [<Item: Item object>, <Item: Item object>, <Item: Item
object>]
```

我们的对象需要一个更好的字符串表示形式。下面再添加一个单元测试：

> 如果已经有测试失败，还要再添加更多的失败测试，通常都要三思而后行，因为这么做会让测试的输出变得更复杂，而且往往你都会担心："还能回到正常运行的状态吗？"这里，测试都很简单，所以我不担忧。

lists/tests/test_models.py (ch13l008)

```
def test_string_representation(self):
    item = Item(text='some text')
    self.assertEqual(str(item), 'some text')
```

测试的结果为：

```
AssertionError: 'Item object' != 'some text'
```

连同另外两个失败，现在开始一并解决：

lists/models.py (ch09l034)

```
class Item(models.Model):
    [...]

    def __str__(self):
        return self.text
```

> 在 Python 2.*x* 的 Django 版本中，字符串表示形式使用 `__unicode__` 方法定制。和很多字符串处理方式一样，Python 3 对此做了简化。参见文档（https://docs.djangoproject.com/en/1.11/topics/python3/#str-and-unicode-methods）。

现在只剩两个失败测试了，而且排序测试的失败消息更易读了：

```
AssertionError: <QuerySet [<Item: i1>, <Item: item 2>, <Item: 3>]> != [<Item:
i1>, <Item: item 2>, <Item: 3>]
```

可以在 `class Meta` 中解决这个问题：

lists/models.py (ch09l035)

```
    class Meta:
        ordering = ('id',)
        unique_together = ('list', 'text')
```

这么做有用吗？

```
AssertionError: <QuerySet [<Item: i1>, <Item: item 2>, <Item: 3>]> != [<Item:
i1>, <Item: item 2>, <Item: 3>]
```

呃，确实有用，从测试结果中可以看到，顺序是一样的，只不过测试没分清。其实我一直会遇到这个问题，因为 Django 中的查询集合不能和列表正确比较。可以在测试中把查询集合转换成列表[1] 解决这个问题：

lists/tests/test_models.py (ch09l036)

```
        self.assertEqual(
            list(Item.objects.all()),
            [item1, item2, item3]
        )
```

这样就可以了，整个测试组件都能通过：

```
OK
```

15.1.3　重写旧模型测试

虽然冗长的模型测试无意间帮我们发现了一个问题，但现在要重写模型测试。重写的过程中我会讲得很详细，因为借此机会要介绍 Django ORM。既然我们已经编写了专门测试排序的测试，现在就可以使用一些较短的测试达到相同的覆盖度。删除 `test_saving_and_retrieving_items`，换成：

lists/tests/test_models.py (ch13l010)

```
class ListAndItemModelsTest(TestCase):

    def test_default_text(self):
        item = Item()
        self.assertEqual(item.text, '')


    def test_item_is_related_to_list(self):
        list_ = List.objects.create()
        item = Item()
        item.list = list_
        item.save()
        self.assertIn(item, list_.item_set.all())

    [...]
```

这么改绰绰有余。初始化一个全新的模型对象，检查属性的默认值，这么做足以确认 models.py 中是否正确设定了一些字段。`test_item_is_related_to_list` 其实是双重保险，确认外键关联是否正常。

借此机会，还要把这个文件中的内容分成专门针对 `Item` 和 `List` 的测试（后者只有一个测试方法，即 `test_get_absolute_url`）：

注 1：也可以考虑使用 `unittest` 中的 `assertSequenceEqual`，以及 Django 测试工具中的 `assertQuerysetEqual`。不过我承认，之前我并没搞清楚怎么使用 `assertQuerysetEqual`。

lists/tests/test_models.py (ch13l011)

```
class ItemModelTest(TestCase):

    def test_default_text(self):
        [...]


class ListModelTest(TestCase):

    def test_get_absolute_url(self):
        [...]
```

修改之后代码更整洁。测试结果如下：

```
$ python manage.py test lists
[...]
Ran 29 tests in 0.092s

OK
```

15.1.4　保存时确实会显示完整性错误

在继续之前还有一个题外话要说。我在第 13 章提到过，保存数据时会出现一些数据完整性错误，还记得吗？是否出现完整性错误完全取决于完整性约束是否由数据库执行。

执行 `makemigrations` 命令试试，你会看到，Django 除了把 `unique_together` 作为应用层约束之外，还想把它加到数据库中：

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0005_auto_20140414_2038.py
    - Change Meta options on item
    - Alter unique_together for item (1 constraint(s))
```

现在，修改检查重复待办事项的测试，把 `.full_clean` 改成 `.save`：

lists/tests/test_models.py

```
    def test_duplicate_items_are_invalid(self):
        list_ = List.objects.create()
        Item.objects.create(list=list_, text='bla')
        with self.assertRaises(ValidationError):
            item = Item(list=list_, text='bla')
            # item.full_clean()
            item.save()
```

测试的结果为：

```
ERROR: test_duplicate_items_are_invalid (lists.tests.test_models.ItemModelTest)
[...]
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

```
[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

可以看出，错误是由 SQLite 导致的，而且错误类型也和我们期望的不一样，我们想得到的是 `ValidationError`，实际却是 `IntegrityError`。

把改动改回去，让测试全部通过：

```
$ python manage.py test lists
[...]
Ran 29 tests in 0.092s
OK
```

然后提交对模型层的修改：

```
$ git status # 会看到改动了测试和模型，还有一个新迁移文件
# 我们给新迁移文件起一个更好的名字
$ mv lists/migrations/0005_auto* lists/migrations/0005_list_item_unique_together.py
$ git add lists
$ git diff --staged
$ git commit -am "Implement duplicate item validation at model layer"
```

15.2　在视图层试验待办事项重复验证

运行功能测试，看看现在我们进展到哪里了：

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
```

运行功能测试时浏览器窗口一闪而过，你可能没看到网站现在处于 500 状态之中。[2] 简单的修改视图层单元测试应该能解决这个问题：

lists/tests/test_views.py (ch13l014)

```
class ListViewTest(TestCase):
    [...]

    def test_for_invalid_input_shows_error_on_page(self):
        [...]


    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        list1 = List.objects.create()
        item1 = Item.objects.create(list=list1, text='textey')
        response = self.client.post(
            f'/lists/{list1.id}/',
            data={'text': 'textey'}
        )
```

注 2：显示一个服务器错误，响应码为 500。你要明白这些术语的意思。

```
        expected_error = escape("You've already got this in your list")
        self.assertContains(response, expected_error)
        self.assertTemplateUsed(response, 'list.html')
        self.assertEqual(Item.objects.all().count(), 1)
```

测试结果为：

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

我们不想让测试出现完整性错误！理想情况下，我们希望在尝试保存数据之前调用 `is_valid` 时，已经注意到有重复。不过在此之前，表单必须知道待办事项属于哪个清单。

现在暂时为这个测试加上 `@skip` 修饰器：

lists/tests/test_views.py (ch13l015)

```
from unittest import skip
[...]

    @skip
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
```

15.3　处理唯一性验证的复杂表单

新建清单的表单只需知道一件事，即新待办事项的文本。为了验证清单中的代办事项是否唯一，表单需要知道使用哪个清单以及待办事项的文本。就像前面我们在 `ItemForm` 类中定义 `save` 方法一样，这一次要重定义表单的构造方法，让它知道待办事项属于哪个清单。

复制前一个表单的测试，稍微做些修改：

lists/tests/test_forms.py (ch13l016)

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm
)
[...]

class ExistingListItemFormTest(TestCase):

    def test_form_renders_item_text_input(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_)
        self.assertIn('placeholder="Enter a to-do item"', form.as_p())


    def test_form_validation_for_blank_items(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_, data={'text': ''})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])
```

```
    def test_form_validation_for_duplicate_items(self):
        list_ = List.objects.create()
        Item.objects.create(list=list_, text='no twins!')
        form = ExistingListItemForm(for_list=list_, data={'text': 'no twins!'})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [DUPLICATE_ITEM_ERROR])
```

要历经几次 TDD 循环，最后才能得到一个自定义的构造方法。这个构造方法会忽略 for_list 参数。（我不会写出全部过程，但我相信你会做完的，对吗？记住，测试山羊能看到一切。）

lists/forms.py (ch09l071)

```
DUPLICATE_ITEM_ERROR = "You've already got this in your list"
[...]
class ExistingListItemForm(forms.models.ModelForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

现阶段的错误应该是：

```
ValueError: ModelForm has no model class specified.
```

接下来，让这个表单继承现有的表单，看测试能不能通过：

lists/forms.py (ch09l072)

```
class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

能通过，现在只剩一个失败测试了：

```
FAIL: test_form_validation_for_duplicate_items
(lists.tests.test_forms.ExistingListItemFormTest)
    self.assertFalse(form.is_valid())
AssertionError: True is not false
```

下面这一步需要了解一点 Django 内部运作机制，你可以阅读 Django 文档中对模型验证和表单验证的介绍。

Django 在表单和模型中都会调用 validate_unique 方法，借助 instance 属性在表单的 validate_unique 方法中调用模型的 validate_unique 方法：

lists/forms.py

```
from django.core.exceptions import ValidationError
[...]

class ExistingListItemForm(ItemForm):

    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.instance.list = for_list
```

```
    def validate_unique(self):
        try:
            self.instance.validate_unique()
        except ValidationError as e:
            e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
            self._update_errors(e)
```

这段代码用到了一点 Django 魔法，先获取验证错误，修改错误消息之后再把错误传回表单。

任务完成，做个简单的提交：

```
$ git diff
$ git commit -a
```

15.4 在清单视图中使用ExistingListItemForm

现在看一下能否在视图中使用这个表单。

删掉测试方法的 `@skip` 修饰器，同时使用前一节创建的常量清理测试。

lists/tests/test_views.py (ch13l049)

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm,
)
[...]

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        [...]
        expected_error = escape(DUPLICATE_ITEM_ERROR)
```

修改之后完整性错误又出现了：

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

解决方法是使用前一节定义的表单类。在此之前，先找到检查表单类的测试，然后按照下面的方式修改：

lists/tests/test_views.py (ch13l050)

```
class ListViewTest(TestCase):
[...]

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
        self.assertContains(response, 'name="text"')
```

```
    [...]

    def test_for_invalid_input_passes_form_to_template(self):
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
```

修改之后测试的结果为：

```
AssertionError: <ItemForm bound=False, valid=False, fields=(text)> is not an
instance of <class 'lists.forms.ExistingListItemForm'>
```

那么就可以修改视图了：

lists/views.py (ch13l051)

```
from lists.forms import ExistingListItemForm, ItemForm
[...]
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
            [...]
```

问题**几乎**都解决了，但又出现了一个意料之外的失败：

```
TypeError: save() missing 1 required positional argument: 'for_list'
```

不再需要使用父类 `ItemForm` 中自定义的 `save` 方法。为此，先编写一个单元测试：

lists/tests/test_forms.py (ch13l053)

```
def test_form_save(self):
    list_ = List.objects.create()
    form = ExistingListItemForm(for_list=list_, data={'text': 'hi'})
    new_item = form.save()
    self.assertEqual(new_item, Item.objects.all()[0])
```

可以让表单调用祖父类中的 `save` 方法：

lists/forms.py (ch13l054)

```
    def save(self):
        return forms.models.ModelForm.save(self)
```

个人观点：这里可以使用 `super`，但是有参数时我选择不用，例如获取祖父类中的方法。我觉得使用 Python 3 的 `super()` 方法获取直接父类很棒，但其他用途太容易出错，而且写出的代码也不好看。你的观点可能与我不同。

搞定！所有单元测试都能通过：

```
$ python manage.py test lists
[...]
Ran 34 tests in 0.082s

OK
```

针对验证的功能测试也能通过：

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
..
 ---------------------------------------------------------------------
Ran 2 tests in 12.048s

OK
```

检查的最后一步——运行**所有**功能测试：

```
$ python manage.py test functional_tests
[...]
.....
 ---------------------------------------------------------------------
Ran 5 tests in 19.048s

OK
```

太棒了，最后还要提交，再回顾一下之前所学的内容。

15.5　小结：目前所学的Django测试知识

我们的应用现在看起来更像是“标准的”Django 应用了，它实现了 Django 常见的三层：模型、表单和视图。测试不再是“试水式”的了，代码也更像是实际应用中应该有的样子了。

每个关键的源码文件都对应一个单元测试文件，下面回顾一下内容最多（层级也最高）的那个，即 test_views（下述代码清单只列出了关键测试和断言）。

如何测试视图

lists/tests/test_views.py

```
class ListViewTest(TestCase):
  def test_uses_list_template(self):
      response = self.client.get(f'/lists/{list_.id}/') ❶
      self.assertTemplateUsed(response, 'list.html') ❷
  def test_passes_correct_list_to_template(self):
      self.assertEqual(response.context['list'], correct_list) ❸
  def test_displays_item_form(self):
      self.assertIsInstance(response.context['form'], ExistingListItemForm) ❹
      self.assertContains(response, 'name="text"')
  def test_displays_only_items_for_that_list(self):
      self.assertContains(response, 'itemey 1') ❺
```

```
            self.assertContains(response, 'itemey 2') ❺
            self.assertNotContains(response, 'other list item 1') ❺
        def test_can_save_a_POST_request_to_an_existing_list(self):
            self.assertEqual(Item.objects.count(), 1) ❻
            self.assertEqual(new_item.text, 'A new item for an existing list') ❻
        def test_POST_redirects_to_list_view(self):
            self.assertRedirects(response, f'/lists/{correct_list.id}/') ❻
        def test_for_invalid_input_nothing_saved_to_db(self):
            self.assertEqual(Item.objects.count(), 0) ❻
        def test_for_invalid_input_renders_list_template(self):
            self.assertEqual(response.status_code, 200)
            self.assertTemplateUsed(response, 'list.html') ❻
        def test_for_invalid_input_passes_form_to_template(self):
            self.assertIsInstance(response.context['form'], ExistingListItemForm) ❼
        def test_for_invalid_input_shows_error_on_page(self):
            self.assertContains(response, escape(EMPTY_ITEM_ERROR)) ❼
        def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
            self.assertContains(response, expected_error)
            self.assertTemplateUsed(response, 'list.html')
            self.assertEqual(Item.objects.all().count(), 1)
```

❶ 使用 Django 测试客户端。

❷ 检查使用的模板。然后在模板的上下文中检查各个待办事项。

❸ 检查每个对象都是希望得到的，或者查询集合中包含正确的待办事项。

❹ 检查表单使用正确的类。

❺ 检查测试模板逻辑：每个 for 和 if 语句都要做最简单的测试。

❻ 对于处理 POST 请求的视图，确保有效和无效两种情况都要测试。

❼ 健全性检查（可选），检查是否渲染指定的表单，而且是否显示错误消息。

为什么要测试这么多？请阅读附录 B。使用基于类的视图重构时，这些测试能保证视图仍然可以正常运行。

接下来要尝试使用一些客户端代码让数据验证更友好。我想你知道这是什么意思。

第16章

试探JavaScript

“如果上帝想让我们享受生活，就不会把他无尽的痛苦当作珍贵的礼物赠与我们。”

——John Calvin[1]

虽然前面实现的验证逻辑很好，但当用户开始修正问题时，让待办事项重复的错误消息消失，就像 HTML5 验证错误那样，不是更好吗？为了达到这个效果，需要使用少量的 JavaScript。

每天使用 Python 这种充满乐趣的语言编程完全把我们宠坏了。JavaScript 是给我们的惩罚。对 Web 开发者而言，这是绕不开的话题。接下来要十分谨慎地试探如何使用 JavaScript。

> 我假设你知道基本的 JavaScript 句法。如果你还没读过《JavaScript 语言精粹》，现在就买一本吧！这本书并不厚。

16.1　从功能测试开始

在 `ItemValidationTest` 类中添加一个新的功能测试：

functional_tests/test_list_item_validation.py (ch14l001)

```
def test_error_messages_are_cleared_on_input(self):
    # 伊迪丝新建一个清单，但方法不当，所以出现了一个验证错误
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys('Banter too thick')
    self.get_item_input_box().send_keys(Keys.ENTER)
```

注 1：*Calvin and the Chipmunk* 中的角色。

```
        self.wait_for_row_in_list_table('1: Banter too thick')
        self.get_item_input_box().send_keys('Banter too thick')
        self.get_item_input_box().send_keys(Keys.ENTER)

        self.wait_for(lambda: self.assertTrue(   ❶
            self.browser.find_element_by_css_selector('.has-error').is_displayed()   ❷
        ))

        # 为了消除错误，她开始在输入框中输入内容
        self.get_item_input_box().send_keys('a')

        # 看到错误消息消失了，她很高兴
        self.wait_for(lambda: self.assertFalse(
            self.browser.find_element_by_css_selector('.has-error').is_displayed()   ❷
        ))
```

❶ 又用到了 `wait_for`，又一次传入 `assertTrue`。

❷ `is_displayed()` 可检查元素是否可见。不能只靠检查元素是否存在于 DOM 中去判断，因为现在要开始隐藏元素了。

无疑，这个测试会失败。但在继续之前，要应用“事不过三，三则重构”原则，因为多次使用 CSS 查找错误消息元素。把这个操作移到一个辅助函数中：

functional_tests/test_list_item_validation.py (ch14l002)

```
class ItemValidationTest(FunctionalTest):

    def get_error_element(self):
        return self.browser.find_element_by_css_selector('.has-error')

    [...]
```

我喜欢把辅助函数放在使用它们的功能测试类中，仅当辅助函数需要在别处使用时才放在基类中，以防止基类太臃肿。这就是 YAGNI 原则。

然后，在 test_list_item_validation.py 中做三次替换，例如：

functional_tests/test_list_item_validation.py (ch14l003)

```
        self.wait_for(lambda: self.assertEqual(
            self.get_error_element().text,
            "You've already got this in your list"
        ))
    [...]
        self.wait_for(lambda: self.assertTrue(
            self.get_error_element().is_displayed()
        ))
    [...]
        self.wait_for(lambda: self.assertFalse(
            self.get_error_element().is_displayed()
        ))
```

得到了一个预期错误：

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
    self.get_error_element().is_displayed()
AssertionError: True is not false
```

可以提交这些代码，作为对功能测试的首次改动。

16.2　安装一个基本的JavaScript测试运行程序

在 Python 和 Django 领域中选择测试工具非常简单。标准库中的 `unittest` 模块完全够用了，而且 Django 测试运行程序也是一个不错的默认选择。除此之外，还有一些替代工具——nose 很受欢迎，Green 是新推出的，我个人对 pytest 的印象比较深刻。不过默认选项很不错，已能满足需求。[2]

在 JavaScript 领域，情况就不一样了。我们在工作中使用 YUI，但我觉得我应该走出去看看有没有其他新推出的工具。我被如此多的选项淹没了——jsUnit、Qunit、Mocha、Chutzpah、Karma、Testacular、Jasmine 等。而且还不仅限于此：选中其中一个工具后（例如 Mocha[3]），我还得选择一个**断言框架**和**报告程序**，或许还要选择一个**模拟技术库**——永远没有终点。

最终，我决定我们应该使用 QUnit，因为它简单，跟 Python 单元测试很像，而且能很好地和 jQuery 配合使用。

在 lists/static 中新建一个目录，将其命名为 tests，把 QUnit JavaScript 和 CSS 两个文件下载到该目录。我们还要在该目录中放入一个 tests.html 文件：

```
$ tree lists/static/tests/
lists/static/tests/
├── qunit-2.0.1.css
├── qunit-2.0.1.js
└── tests.html
```

QUnit 的 HTML 样板文件内容如下，其中包含一个冒烟测试：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Javascript tests</title>
  <link rel="stylesheet" href="qunit-2.0.1.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
```

注 2：无可否认，一旦开始找 Python BDD 工具，情况会稍微复杂一些。

注 3：纯粹是因为 Mocha 提供了 NyanCat 测试运行程序。

```
  <script src="qunit-2.0.1.js"></script>

  <script>

QUnit.test("smoke test", function (assert) {
  assert.equal(1, 1, "Maths works!");
});

  </script>
</body>
</html>
```

仔细分析这个文件时，要注意几个重要的问题：使用第一个 `<script>` 标签引入 qunit-2.0.1，然后在第二个 `<script>` 标签中编写测试的主体。

如果在 Web 浏览器中打开这个文件（不用运行开发服务器，在硬盘中找到这个文件即可），会看到类似图 16-1 所示的页面。

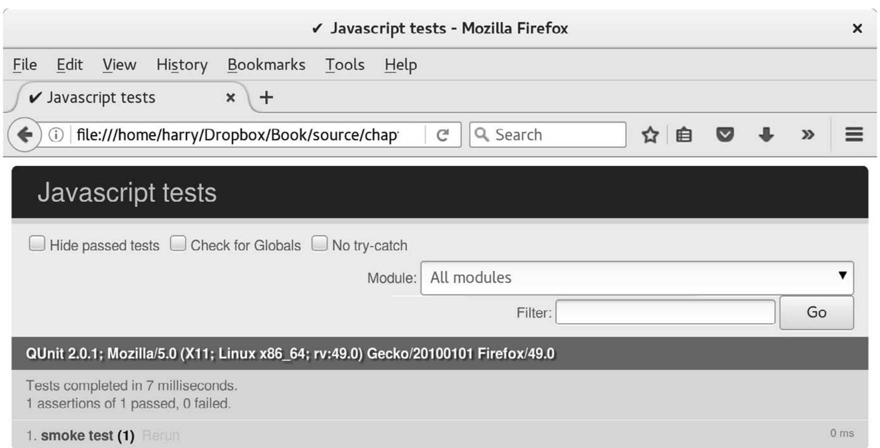


图 16-1：QUnit 的基本界面

查看测试代码会发现，和我们目前编写的 Python 测试有很多相似之处：

```
QUnit.test("smoke test", function (assert) { ❶
    assert.equal(1, 1, "Maths works!"); ❷
});
```

❶ `QUnit.test` 函数定义一个测试用例，有点儿类似 Python 中的 `def test_something(self)`。`test` 函数的第一个参数是测试名，第二个参数是一个函数，定义这个测试的主体。

❷ `assert.equal` 函数是一个断言，和 `assertEqual` 非常像，比较两个参数的值。不过，和在 Python 中不同，不管失败还是通过都会显示消息，所以消息应该使用肯定式而不是否定式。

为什么不修改这些参数，故意让测试失败，看看效果如何呢？

16.3 使用jQuery和<div>固件元素

下面来摸索一下这个测试框架能做什么，并借此开始使用一些 jQuery（不可或缺的库，为操纵 DOM 提供跨浏览器兼容的 API）。

> 如果你从未用过 jQuery，在行文的过程中我会尝试解说，以防你完全不懂。这不是 jQuery 教程，所以在阅读本章的过程中最好抽出一两个小时研究 jQuery。

从 jquery.com 下载新版 jQuery，保存到 lists/static 文件夹中。

下面在测试文件中使用 jQuery，并添加几个 HTML 元素。先试着显示和隐藏元素，并编写几个断言，检查元素的可见性：

lists/static/tests/tests.html

```
    <div id="qunit-fixture"></div>

    <form>  ❶
      <input name="text" />
      <div class="has-error">Error text</div>
    </form>

    <script src="../jquery-3.1.1.min.js"></script>  ❷
    <script src="qunit-2.0.1.js"></script>

    <script>

  QUnit.test("smoke test", function (assert) {
    assert.equal($('.has-error').is(':visible'), true);  ❸❹
    $('.has-error').hide();  ❺
    assert.equal($('.has-error').is(':visible'), false);  ❻
  });

    </script>
```

❶ `<form>` 及其中的内容放在那儿是为了表示真实的清单页面中的内容。

❷ 加载 jQuery。

❸ jQuery 魔法从这里开始！ `$` 是 jQuery 的瑞士军刀，用来查找 DOM 中的内容。`$` 的第一个参数是 CSS 选择符，要查找类为“has-error”的所有元素。查找得到的结果是一个对象，表示一个或多个 DOM 元素。然后，可以在这个对象上使用很多有用的方法处理或者查看这些元素。

❹ 其中一个方法是 `.is`，它的作用是指出某个元素的表现是否和指定的 CSS 属性匹配。这里使用 `:visible` 检查元素是否显示出来。

❺ 使用 jQuery 提供的 `.hide()` 方法隐藏这个 `<div>` 元素。其实，这个方法是在元素上动态设定 `style="display: none"` 属性。

❻ 最后，使用第二个 `assert.equal` 断言检查隐藏是否成功。

刷新浏览器后应该会看到所有测试都通过了。

在浏览器中期望 QUnit 得到的结果如下：

```
2 assertions of 2 passed, 0 failed.
1. smoke test (2)
```

下面要介绍如何使用固件（fixture）。直接复制测试：

lists/static/tests/tests.html

```
  <script>

QUnit.test("smoke test", function (assert) {
  assert.equal($('.has-error').is(':visible'), true);
  $('.has-error').hide();
  assert.equal($('.has-error').is(':visible'), false);
});
QUnit.test("smoke test 2", function (assert) {
  assert.equal($('.has-error').is(':visible'), true);
  $('.has-error').hide();
  assert.equal($('.has-error').is(':visible'), false);
});

  </script>
```

其中一个测试失败了，有点儿出乎预料，如图 16-2 所示。

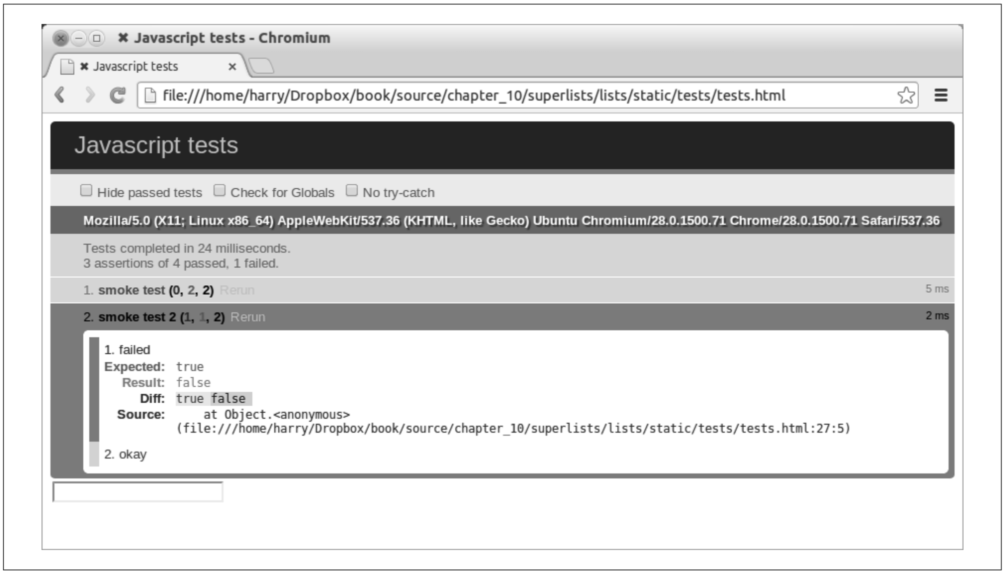


图 16-2：两个测试中有一个失败了

测试失败的原因是，第一个测试把显示错误消息的 `div` 元素隐藏了，所以运行第二个测试时，一开始这个元素就是隐藏的。

> QUnit 中的测试不会按照既定的顺序运行，所以不要觉得第一个测试一定会在第二个测试之前运行。多刷新几次试试，你会发现失败的测试有变化。

我们需要一种方法在测试之间执行清理工作，有点儿类似于 `setUp` 和 `tearDown`，或者像 Django 测试运行程序一样，运行完每个测试后还原数据库。`id` 为 `qunit-fixture` 的 `<div>` 元素就是我们正在寻找的方法。把表单移到这个元素中：

lists/static/tests/tests.html

```
<div id="qunit"></div>
<div id="qunit-fixture">
    <form>
        <input name="text" />
        <div class="has-error">Error text</div>
    </form>
</div>

<script src="../jquery-3.1.1.min.js"></script>
```

你可能已经猜到了，每次运行测试前，jQuery 都会还原这个固件元素中的内容。因此，两个测试都能通过了：

```
4 assertions of 4 passed, 0 failed.
1. smoke test (2)
2. smoke test 2 (2)
```

16.4　为想要实现的功能编写JavaScript单元测试

现在我们已经熟悉这个 JavaScript 测试工具了，所以可以只留下一个测试，开始编写真正的测试代码了：

lists/static/tests/tests.html

```
  <script>

QUnit.test("errors should be hidden on keypress", function (assert) {
  $('input[name="text"]').trigger('keypress'); ❶
  assert.equal($('.has-error').is(':visible'), false);
});

  </script>
```

❶ jQuery 提供的 `.trigger` 方法主要用于测试，作用是在指定的元素上触发一个 JavaScript DOM 事件。这里使用的是 keypress 事件，当用户在指定的输入框中输入内容时，浏览器就会触发这个事件。

> 这里 jQuery 隐藏了很多复杂的细节。不同浏览器之间处理事件的方式大不一样，详情请访问 Quirksmode.org。jQuery 之所以这么受欢迎就是因为它消除了这些差异。

这个测试的结果为：

```
0 assertions of 1 passed, 1 failed.
1. errors should be hidden on keypress (1, 0, 1)
    1. failed
        Expected: false
        Result: true
```

假设我们想把代码放在单独的 JavaScript 文件中，命名为 list.js：

lists/static/tests/tests.html

```
  <script src="../jquery-3.1.1.min.js"></script>
  <script src="../list.js"></script>
  <script src="qunit-2.0.1.js"></script>

  <script>
    [...]
```

若想让这个测试通过，所需的最简代码如下所示：

lists/static/list.js

```
$('.has-error').hide();
```

确实通过了：

```
1 assertions of 1 passed, 0 failed.
1. errors should be hidden on keypress (1)
```

但显然还有个问题，最好再添加一个测试：

lists/static/tests/tests.html

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  assert.equal($('.has-error').is(':visible'), true);
});
```

得到一个预期的失败：

```
1 assertions of 2 passed, 1 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1, 0, 1)
    1. failed
        Expected: true
        Result: false
[...]
```

然后，可以使用一种更真实的实现方式：

lists/static/list.js

```
$('input[name="text"]').on('keypress', function () {   ❶
  $('.has-error').hide();
});
```

❶ 这行代码的意思是：查找所有 `name` 属性为“text”的 `input` 元素，然后在找到的每个元素上附属一个事件监听器，作用在 keypress 事件**上**。事件监听器是那个行间函数，其作用是隐藏类为 `.has-error` 的所有元素。

这样能让测试通过吗？不能。

```
1 assertions of 2 passed, 1 failed.
1. errors should be hidden on keypress (1, 0, 1)
    1. failed
        Expected: false
        Result: true
[...]
2. errors aren't hidden if there is no keypress (1)
```

可恶！这是为什么呢？

16.5　固件、执行顺序和全局状态：JavaScript 测试的重大挑战

一般来说，JavaScript 的一大难点，尤其对测试而言，就是理解代码的执行顺序（何时发生什么）。我们想知道 list.js 中的代码何时运行，每个测试何时运行；我们还想知道代码的运行对全局状态（网页的 DOM）有何影响，以及每次测试后是如何清理固件的。

16.5.1　使用console.log打印调试信息

下面在测试中添加几个 `console.log`，打印调试信息：

lists/static/tests/tests.html

```
  <script>

console.log('qunit tests start');

QUnit.test("errors should be hidden on keypress", function (assert) {
  console.log('in test 1');
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  console.log('in test 2');
  assert.equal($('.has-error').is(':visible'), true);
});
  </script>
```

然后在 JavaScript 代码中也这样做：

lists/static/list.js (ch14l015)

```
$('input[name="text"]').on('keypress', function () {
  console.log('in keypress handler');
  $('.has-error').hide();
});
console.log('list.js loaded');
```

运行测试，打开浏览器的调试控制台（通常可按 Ctrl-Shift-I 组合键），你应该会看到类似图 16-3 所示的输出。

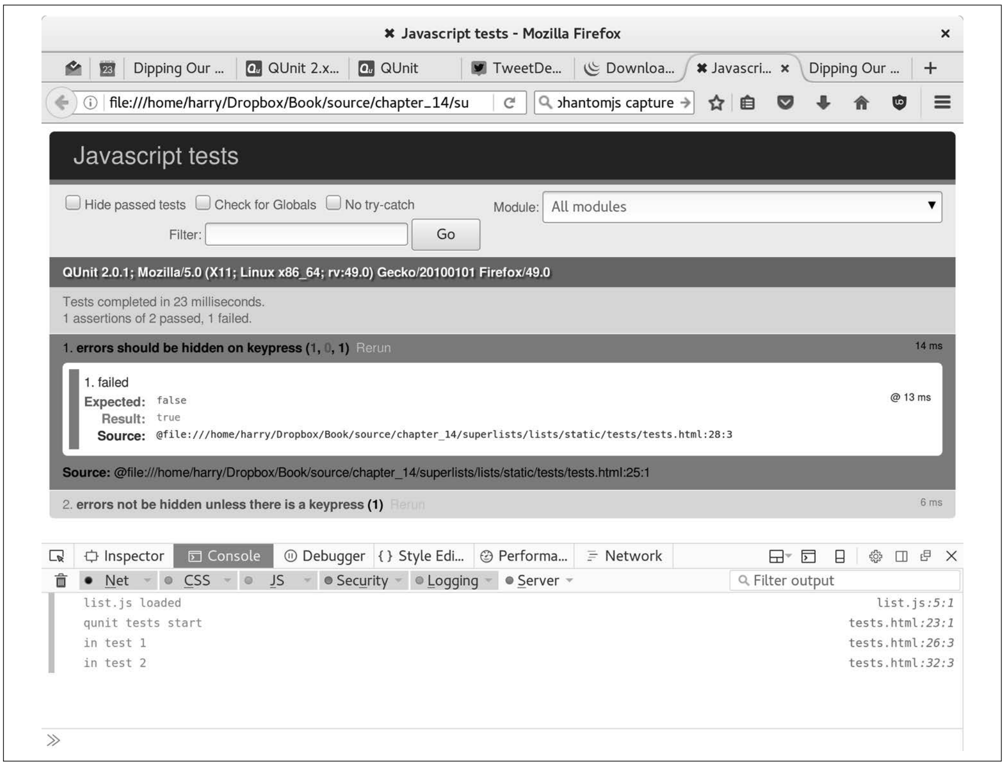


图 16-3：在 QUnit 测试中使用 `console.log` 输出的调试信息

我们看到了什么？

- list.js 先被加载，因此事件监听器应该依附到输入元素上了。
- 然后加载 QUnit 测试文件。
- 最后运行各个测试。

但仔细一想，每个测试都会“还原”固件 `div`，也就是销毁输入元素后再重建。因此，每次运行测试时，list.js 看到并依附事件监听器的输入元素都是全新的。

16.5.2　使用初始化函数精确控制执行时间

我们需要进一步掌控 JavaScript 的执行顺序，而不是依赖 `<script>` 标签加载并运行 list.js 中的代码。为此，常见的做法是定义“初始化”函数，在测试（以及真实的场景）中需要的地方调用：

lists/static/list.js

```
var initialize = function () {
  console.log('initialize called');
  $('input[name="text"]').on('keypress', function () {
    console.log('in keypress handler');
    $('.has-error').hide();
  });
};
console.log('list.js loaded');
```

然后在测试文件中的每个测试中调用 `initialize`：

lists/static/tests/tests.html (ch14l017)

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  console.log('in test 1');
  initialize();
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  console.log('in test 2');
  initialize();
  assert.equal($('.has-error').is(':visible'), true);
});
```

现在，测试能通过，而且调试输出更合理了：

```
2 assertions of 2 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)

list.js loaded
qunit tests start
in test 1
initialize called
in keypress handler
in test 2
initialize called
```

太棒了！下面把 `console.log` 删掉：

lists/static/list.js

```
var initialize = function () {
  $('input[name="text"]').on('keypress', function () {
    $('.has-error').hide();
  });
};
```

把测试中的也删掉：

lists/static/tests/tests.html

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  initialize();
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  initialize();
  assert.equal($('.has-error').is(':visible'), true);
});
```

关键时刻到了。现在引入 jQuery 和我们的脚本，在真实的页面中调用初始化函数：

lists/templates/base.html (ch14l020)

```
    </div>
    <script src="/static/jquery-3.1.1.min.js"></script>
    <script src="/static/list.js"></script>

    <script>
      initialize();
    </script>

  </body>
</html>
```

习惯做法是在 HTML 的 `body` 元素末尾引入脚本；这样的话，用户无须等到所有 JavaScript 都加载完就能看到页面中的内容。此外，还能保证运行脚本前加载了大部分 DOM。

然后运行功能测试：

```
$ python manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input
[...]

Ran 1 test in 3.023s

OK
```

太棒了！做次提交：

```
$ git add lists/static
$ git commit -m "add jquery, qunit tests, list.js with keypress listeners"
```

16.6　经验做法：onload样板代码和命名空间

哦，还有一件事。`initialize` 这个函数名称太普通了，如果引入的某个第三方 JavaScript 工具也有名为 `initialize` 的函数呢？下面为其添加别人不太可能使用的“命名空间”：

lists/static/list.js

```
window.Superlists = {};  ❶
window.Superlists.initialize = function () {   ❷
  $('input[name="text"]').on('keypress', function () {
    $('.has-error').hide();
  });
};
```

❶ 声明一个对象，作为“window”全局对象的属性，为其起一个别人不太可能使用的名称。

❷ 然后把 `initialize` 函数设为命名空间中那个对象的属性。

> 若想在 JavaScript 中处理命名空间，还有很多更取巧的方式，不过都太复杂，而且我也不是专家，无法一一说明。如果想深入学习，请搜索 require.js——这似乎已经成为了标准做法，至少在当下是这样。

lists/static/tests/tests.html

```
  <script>
QUnit.test("errors should be hidden on keypress", function (assert) {
  window.Superlists.initialize();
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  window.Superlists.initialize();
  assert.equal($('.has-error').is(':visible'), true);
});
  </script>
```

最后，如果 JavaScript 需要和 DOM 交互，最好把相应的代码包含在 `onload` 样板代码中，确保在执行脚本之前完全加载了页面。目前的做法也能正常运行，因为 `<script>` 标签在页面底部，但不能依赖这种方式。

jQuery 提供的 `onload` 样板代码非常简洁：

lists/templates/base.html

```
    <script>

$(document).ready(function () {
  window.Superlists.initialize();
});

    </script>
```

更多信息请阅读 jQuery `.ready()` 的文档。

16.7　JavaScript测试在TDD循环中的位置

你可能想知道 JavaScript 测试在双重 TDD 循环中处于什么位置。答案是，JavaScript 测试和 Python 单元测试扮演的角色完全相同。

(1)　编写一个功能测试，看着它失败。
(2)　判断接下来需要哪种代码，Python 还是 JavaScript ？
(3)　使用选中的语言编写单元测试，看着它失败。
(4)　使用选中的语言编写一些代码，让测试通过。
(5)　重复上述步骤。

> 想多练习使用 JavaScript 吗？当用户在输入框内点击或者输入内容时，看看你能否隐藏错误消息。实现的过程中应该还要编写功能测试。

我们几乎可以进入第三部分了，但在此之前还有最后一步：把修改后的新代码部署到服务器中。别忘了最后再提交一次（包含 base.html）！

16.8　一些缺憾

本章的目的是介绍 JavaScript 测试基础知识，并说明 JavaScript 测试在 TDD 循环中的位置。下面几点可做深入研究。

- 目前的测试只检查 JavaScript 能否在一个页面中使用。JavaScript 之所以能使用，是因为在 base.html 中引入了 JavaScript 文件。如果只在 home.html 中引入 JavaScript 文件，测试也能通过。你可以选择在哪个文件中引入，但也可以再编写一个测试。
- 编写 JavaScript 时，应该尽量利用编辑器提供的协助，避免常见的问题。试一下句法 / 错误检查工具（linter），例如 jslint 和 jshint。
- QUnit 希望你在真正的 Web 浏览器中“运行”测试，这样有利于创建与网站的真实内容匹配的 HTML 固件，供测试使用。但是，JavaScript 测试也能在命令行中运行。第 24 章有个例子。
- 前端开发圈目前流行 angular.js 和 React 这样的 MVC 框架。这些框架的教程大都使用一个 RSpec 式断言库，名为 Jasmine。如果你想使用 MVC 框架，使用 Jasmine 比 QUnit 更方便。

本书后面还会涉及 JavaScript。如果你有兴趣，可以翻阅附录 F 的内容。

JavaScript 测试笔记

- Selenium 最大的优势之一是可以测试 JavaScript 是否真的能使用，就像测试 Python 代码一样。
- JavaScript 测试运行库有很多，QUnit 和 jQuery 联系紧密，这是我选择使用它的主要原因。
- 不管使用哪个测试库，都要设法解决 JavaScript 测试面对的主要挑战：**管理全局状态**。这包括：
  - ♦ DOM/HTML 固件；
  - ♦ 命名空间；
  - ♦ 理解并控制执行顺序。
- 我说 JavaScript 很糟糕并不是出于真心。JavaScript 其实也可以很有趣。不过我还是要再说一次：一定要阅读《JavaScript 语言精粹》。

第 17 章

部署新代码

现在可以把全新的验证代码部署到线上服务器了。借此机会，我们也能再次在实践中使用自动化部署脚本。

> 此刻，我由衷地感谢 Andrew Godwin 和整个 Django 团队。在 Django 1.7 之前，我写了很长一节内容，专门说明如何迁移。现在，因为迁移可以自动执行，所以那整节内容都不需要了。感谢你们的辛勤工作。

17.1　部署到过渡服务器

先部署到过渡服务器中：

```
$ git push
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
Disconnecting from superlists-staging.ottg.eu... done.
```

重启 Gunicorn：

```
elspeth@server:$ sudo systemctl restart gunicorn-superlists-staging.ottg.eu
```

然后在过渡服务器中运行测试：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
OK
```

17.2　部署到线上服务器

假设在过渡服务器上一切正常，那么就可以运行脚本，部署到线上服务器：

```
$ fab deploy:host=elspeth@superlists.ottg.eu

elspeth@server:$ sudo service gunicorn-superlists.ottg.eu restart
```

17.3　如果看到数据库错误该怎么办

迁移中引入了一个完整性约束，你可能会发现迁移执行失败，因为某些现有的数据违背了约束规则。

此时有两个选择。

- 删除服务器中的数据库，然后再部署试试。毕竟这只是一个小项目！
- 或者，学习如何迁移数据（参见附录 D）。

17.4　总结：为这次新发布打上Git标签

最后要做的一件事是，在 VCS 中为这次发布打上标签——始终能跟踪线上运行的是哪个版本十分重要：

```
$ git tag -f LIVE # 需要指定-f，因为我们要替换旧标签
$ export TAG=`date +DEPLOYED-%F/%H%M`
$ git tag $TAG
$ git push -f origin LIVE $TAG
```

有些人不喜欢使用 `push -f`，也不喜欢更新现有的标签，而是使用某种版本号标记每次发布。你觉得哪种方法好就用哪种。

至此，第二部分结束了。接下来我们要进入第三部分，介绍更让人兴奋的话题。真令人期待啊！

部署过程回顾

目前我们部署过好几次了，下面回顾一下整个过程。

- 执行 `git push` 命令，推送最新的代码。
- 部署到过渡服务器，运行功能测试。
- 部署到线上服务器。
- 为最新的版本打上标签。

随着项目的增长，部署过程会变得越来越复杂。如果没有好的自动化方案，部署将变得难以维护，处处都需要自己动手检查和操作。这个话题还有很多内容，不过已经超出本书范畴。记得阅读附录 C，另外再研究一下“持续部署”。

第三部分
高级话题

“噢，天呐，什么，还有一部分？哈利，我累了，已经看了两百多页，觉得无法再看完另一部分了，而且这一部分还是‘高级’话题……我能不能跳过这一部分呢？”

噢，不，你不能。这一部分虽然被称为“高级”话题，但其实有很多对 TDD 和 Web 开发十分重要的知识，因此不能跳过。这一部分甚至比前两部分**还重要**。

我们会讨论如何集成和测试第三方系统。重用现有的组件对现代 Web 开发十分重要。还会介绍模拟技术和测试隔离，这两种技术是 TDD 的核心，而且在最简单的代码基中也会用到。最后要讨论服务器端调试技术和测试固件，以及如何搭建持续集成（Continuous Integration）环境。这些技术在项目中并不是可有可无的奢侈附属品，它们其实都很重要。

这一部分的学习曲线难免稍微陡峭一些。你可能要多读几遍才能领会，或者第一次操作时可能无法正常运行，需要自己调试一番。但要坚持下去，知识越难，只要学会了，收获也就越大。如果你遇到难题，我十分乐意帮忙，请给我发电子邮件，地址是 obeythetestinggoat@gmail.com。

来吧，我保证最重要的知识就在这一部分！

第 18 章

用户身份验证、探究及去掉探究代码

精美的待办事项清单网站上线好几天了，用户开始提供反馈。他们说：“我们喜欢这个网站，但总是找不到使用过的清单，又很难靠死记硬背来记住 URL。如果网站能记住我们创建过哪些清单就好了。”

还记得亨利 · 福特说过的那句“快马”名言吗？[1]收到用户的反馈后一定要深入分析并思考：“用户真正需要的是什么？满足用户的需求时如何使用自己一直想尝试的酷炫新技术？”

很明显，这些用户需要的是某种用户账户系统。那么就直接实现认证功能吧。

我们不会费力气自己存储密码，这是 20 世纪 90 年代的技术，而且存储用户密码还是个安全噩梦，所以还是交给第三方去完成吧。我们要使用一种称为无密码验证的技术。

（如果你**坚持**要自己存储密码，可以使用 Django 提供的 `auth` 模块。这个模块很友好也很简单，具体的实现方法留给你自己去发掘。）

18.1　无密码验证

为了不自己存储密码，我们要使用什么身份验证系统呢？ OAuth、OpenID，还是“通过 Facebook 登录”？对我来说，这些认证系统都有让人无法接受的缺点——为什么要让 Google 或 Facebook 知道我何时登录过什么网站呢？

本书第 1 版使用的是一个叫“Persona”的实验性项目，这个项目由 Mozilla 一些具有理想主义嬉皮士精神的技术人员研发。但可惜的是，它如今已被废弃。

注 1：亨利 · 福特是福特汽车公司的创始人。这里所说的“快马”名言全句是：“If I had asked people what they wanted, they would have said faster horses.”但也有人说福特并没说过这句话。——译者注

但我找到了一个不错的替代方案，这种身份验证方式叫作“无密码验证”，你也可以称之为“只用电子邮件验证”。

发明这个系统的人觉得为每个网站都创建密码很麻烦，而且他发现他使用的密码都是随机的，用完就“扔”，根本不会尝试去记，等到再需要登录时使用“忘记密码”功能就好。详情请参阅 Medium 上的文章名为“Passwords are Obselete”。

无密码验证系统的原理是，只使用电子邮件地址确认身份。既然你希望使用“忘记密码”功能，就说明你信任电子邮件地址，那为什么不直捣黄龙呢？如果用户想登录，就生成一个唯一的 URL，通过电子邮件发给用户，用户点击 URL 后即可登录网站。

世界上没有完美的系统，为了给线上网站提供好的登录方案，需要深思熟虑的细节有很多。但这是个实验性项目，不必太费心。

18.2 探索性编程（又名“探究”）

在撰写本章之前，我对无密码验证的认识只限于前文给出的那篇文章中的简要说明。我没见过具体的代码，也不知道应该从哪入手。

我们在第 13、14 章中见识到，可以使用单元测试探索新 API 的用法。但有时你不想写测试，只是想捣鼓一下，看 API 是否能用，目的是学习和领会。这么做绝对可行。学习新工具，或者研究新的可行性方案时，一般都可以适当地把严格的 TDD 流程放在一边，不编写测试或编写少量的测试，先把基本的原型开发出来。测试山羊并不介意暂时睁一只眼闭一只眼。

这种创建原型的过程一般叫作“探究”（spike）。这么叫的原因众所周知。

首先，我研究了现有的 Python 和 Django 身份验证码，比如 django-allauth 和 python-social-auth，但这两个包目前都太过复杂。（回头一想，自己编程多有趣！）

所以，我决定亲自动手，经过一番潜心研究之后，终于写出了刚好能用的代码。下面我要向你展示我的实现过程，然后再过一遍，去掉探索性代码，即把原型替换成经过测试、可在生产环境使用的代码。

你应该自己动手，把这些代码添加到自己的网站中，这样才能得到试验的对象。然后使用自己的电子邮件地址登录试试，证明代码确实可用。

18.2.1 为此次探究新建一个分支

着手探究之前，最好新建一个分支，这样就不用担心探究过程中提交的代码把 VCS 中的生产代码搞乱了：

```
$ git checkout -b passwordless-spike
```

下面在便签上记下希望从这次探究中学到的东西。



18.2.2　前端登录UI

先从前端入手。在导航栏中放一个表单，让用户输入电子邮件地址，并为通过身份验证的用户提供退出链接：

lists/templates/base.html (ch16l001)

```
<body>
  <div class="container">

    <div class="navbar">
      {% if user.is_authenticated %}
        <p>Logged in as {{ user.email}}</p>
        <p><a id="id_logout" href="{% url 'logout' %}">Log out</a></p>
      {% else %}
        <form method="POST" action ="{% url 'send_login_email' %}">
          Enter email to log in: <input name="email" type="text" />
          {% csrf_token %}
        </form>
      {% endif %}
    </div>

    <div class="row">
    [...]
```

18.2.3　从Django中发出邮件

登录过程是这样的。

- 有人想登录时，就生成一个唯一的秘密令牌，存储在数据库中，并与他的电子邮件地址关联起来，然后把令牌发给那个人。
- 他查看电子邮件，里面有个包含令牌的 URL。
- 他点击链接后，我们检查令牌是否存在于数据库中，如果存在就登入相应的用户。

首先，为账户准备一个应用：

```
$ python manage.py startapp accounts
```

然后在 urls.py 中至少设置一个 URL。先是位于顶级的 superlists/urls.py 文件。

superlists/urls.py (ch16l003)

```
from django.conf.urls import include, url
from lists import views as list_views
from lists import urls as list_urls
from accounts import urls as accounts_urls

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)),
    url(r'^accounts/', include(accounts_urls)),
]
```

然后是 accounts 模块中的 urls.py 文件：

accounts/urls.py (ch16l004)

```
from django.conf.urls import url
from accounts import views

urlpatterns = [
    url(r'^send_email$', views.send_login_email, name='send_login_email'),
]
```

下述视图负责创建与用户在登录表单中输入的电子邮件地址关联的令牌：

accounts/views.py (ch16l005)

```
import uuid
import sys
from django.shortcuts import render
from django.core.mail import send_mail

from accounts.models import Token


def send_login_email(request):
    email = request.POST['email']
    uid = str(uuid.uuid4())
    Token.objects.create(email=email, uid=uid)
    print('saving uid', uid, 'for email', email, file=sys.stderr)
    url = request.build_absolute_uri(f'/accounts/login?uid={uid}')
    send_mail(
        'Your login link for Superlists',
        f'Use this link to log in:\n\n{url}',
        'noreply@superlists',
        [email],
    )
    return render(request, 'login_email_sent.html')
```

为此，我们要显示一条消息，指明电子邮件已经发出：

accounts/templates/login_email_sent.html (ch16l006)

```
<html>
<h1>Email sent</h1>

<p>Check your email, you'll find a message with a link that will log you into
the site.</p>

</html>
```

（这段代码只是临时的，实际使用中应该集成到 base.html 模板里。）

为了让 Django 的 `send_mail` 函数工作，更重要的是告诉 Django 我们的电子邮件服务器地址。我暂时先使用自己的 Gmail 账户。[2] 你可以使用任何你想用的电子邮件服务提供商，只要支持 SMTP 就行：

superlists/settings.py (ch16l007)

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'obeythetestinggoat@gmail.com'
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASSWORD')
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

如果你也想使用 Gmail，可能需要访问 Google 账户的安全设置页面。如果你在使用双因素身份验证，可能需要设置应用专用的密码；如果没用双因素身份验证，可能也要允许较不安全的应用访问。鉴于此，你可以新建一个 Google 账户，而不使用包含敏感数据的账户。

18.2.4　使用环境变量，避免源码中出现机密信息

每个项目终究都要以一种方式处理“机密信息”——像电子邮件密码或 API 密钥这类不想和整个世界分享的数据。如果你的仓库是私有的，或许还可以存储在 Git 中，但事实往往不是这样。而且这还涉及在开发和生产环境中使用不同的设置。（还记得在第 11 章中我们是如何处理 Django 的 `SECRET_KEY` 设置的吗？）

这种配置通常使用环境变量存储，上述代码中的 `os.environ.get` 就是用于读取环境变量的。

为此，我要在运行开发服务器的 shell 中设定环境变量：

```
$ export EMAIL_PASSWORD="sekrit"
```

后面还会在过渡服务器中添加这个环境变量。

注 2：我刚刚是不是说了一大堆关于使用 Google 登录对隐私有多大影响的话？那为什么现在又要使用 Gmail 呢？是的，这的确相互矛盾。（老实讲，有一天我会抛弃 Gmail 的！）但这里只是用于测试，我又没强制用户使用 Google。

18.2.5　在数据库中存储令牌

情况进展如何？



为了把令牌存储在数据库中，我们需要一个模型。这个模型要把电子邮件地址与唯一的 ID 关联起来——这没什么难的：

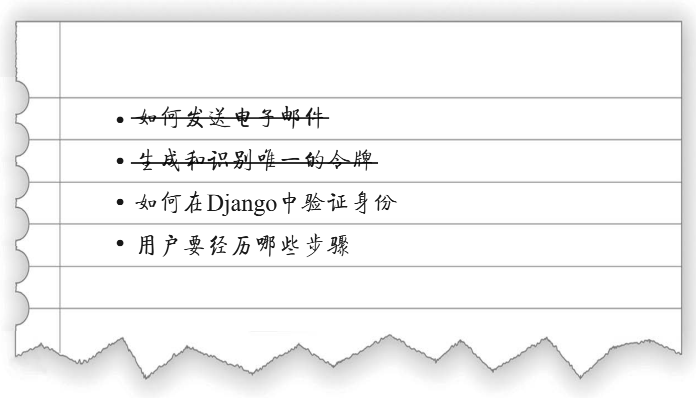
accounts/models.py (ch16l008)

```
from django.db import models

class Token(models.Model):
    email = models.EmailField()
    uid = models.CharField(max_length=255)
```

18.2.6　自定义身份验证模型

既然已经谈到模型了，那就试验一下如何在 Django 中验证身份吧。



首先，要有一个用户模型。在我刚开始编写时，自定义用户模型还是 Django 的新特性，所以我仔细研读了 Django 的身份验证文档，努力找出最简单的方法：

accounts/models.py (ch16l009)

```
[...]
from django.contrib.auth.models import (
    AbstractBaseUser, BaseUserManager, PermissionsMixin
)

class ListUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(primary_key=True)
    USERNAME_FIELD = 'email'
    #REQUIRED_FIELDS = ['email', 'height']

    objects = ListUserManager()

    @property
    def is_staff(self):
        return self.email == 'harry.percival@example.com'

    @property
    def is_active(self):
        return True
```

这算是一个极简用户模型，它只有一个字段，没有名字、姓氏或用户名之类的，尤其是没有密码字段。我们不必为此操心！

但我要再次说明，这段代码不适合在生产环境使用，里面注释掉了一行代码，还硬编码了我的电子邮件地址。去掉试探代码时会做大幅调整。

此外，还要为用户模型提供一个模型管理器：

accounts/models.py (ch16l010)

```
[...]
class ListUserManager(BaseUserManager):

    def create_user(self, email):
        ListUser.objects.create(email=email)

    def create_superuser(self, email, password):
        self.create_user(email)
```

现阶段你不用管模型管理器是什么，目前就是需要这么一个东西，有了它才能工作。去掉试探代码时，我们会分析每一段代码，确定哪些是可以在生产环境使用的，彻底弄明白这些代码的作用。

18.2.7　结束自定义Django身份验证功能

就要完成了。最后一步要识别令牌，然后登入用户。做完这一步，便签上记录的事项基本上都可以划掉了。



下面是点击电子邮件中的链接后触发的视图：

accounts/views.py (ch16l011)

```
import uuid
import sys
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.core.mail import send_mail
from django.shortcuts import redirect, render
[...]

def login(request):
    print('login view', file=sys.stderr)
    uid = request.GET.get('uid')
    user = authenticate(uid=uid)
    if user is not None:
        auth_login(request, user)
    return redirect('/')
```

`authenticate` 函数调用 Django 的身份验证框架，我们已经将它配置为使用“自定义的身份验证框架”，作用是验证 UID，返回电子邮件对应的用户。

我们本可以在这个视图中直接结束，但是最好按照 Django 预期的方式组织代码，实现关注点分离：

accounts/authentication.py (ch16l012)

```
import sys
from accounts.models import ListUser, Token

class PasswordlessAuthenticationBackend(object):

    def authenticate(self, uid):
        print('uid', uid, file=sys.stderr)
        if not Token.objects.filter(uid=uid).exists():
            print('no token found', file=sys.stderr)
            return None
```

```
        token = Token.objects.get(uid=uid)
        print('got token', file=sys.stderr)
        try:
            user = ListUser.objects.get(email=token.email)
            print('got user', file=sys.stderr)
            return user
        except ListUser.DoesNotExist:
            print('new user', file=sys.stderr)
            return ListUser.objects.create(email=token.email)

    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

这段代码也不例外，打印了很多调试信息，还有一些重复，不适合在生产环境使用。不过我们现在要的是能用就行。

最后，编写退出视图：

accounts/views.py (ch16l013)

```
from django.contrib.auth import login as auth_login, logout as auth_logout
[...]

def logout(request):
    auth_logout(request)
    return redirect('/')
```

把 `login` 和 `logout` 视图添加到 urls.py 中：

accounts/urls.py (ch16l014)

```
from django.conf.urls import url
from accounts import views

urlpatterns = [
    url(r'^send_email$', views.send_login_email, name='send_login_email'),
    url(r'^login$', views.login, name='login'),
    url(r'^logout$', views.logout, name='logout'),
]
```

坚持住，就快结束了！在 settings.py 中启用 auth 后端和这个 accounts 应用：

superlists/settings.py (ch16l015)

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
    'accounts',
]

AUTH_USER_MODEL = 'accounts.ListUser'
```

```
AUTHENTICATION_BACKENDS = [
    'accounts.authentication.PasswordlessAuthenticationBackend',
]

MIDDLEWARE = [
[...]
```

执行 `makemigrations` 命令，为令牌和用户模型生成迁移：

```
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model ListUser
    - Create model Token
```

再执行 `migrate` 命令构建数据库：

```
$ python manage.py migrate
[...]
Running migrations:
  Applying accounts.0001_initial... OK
```

一切就绪！为什么不执行 `runserver` 命令启动开发服务器，看看实际效果呢（见图 18-1）？

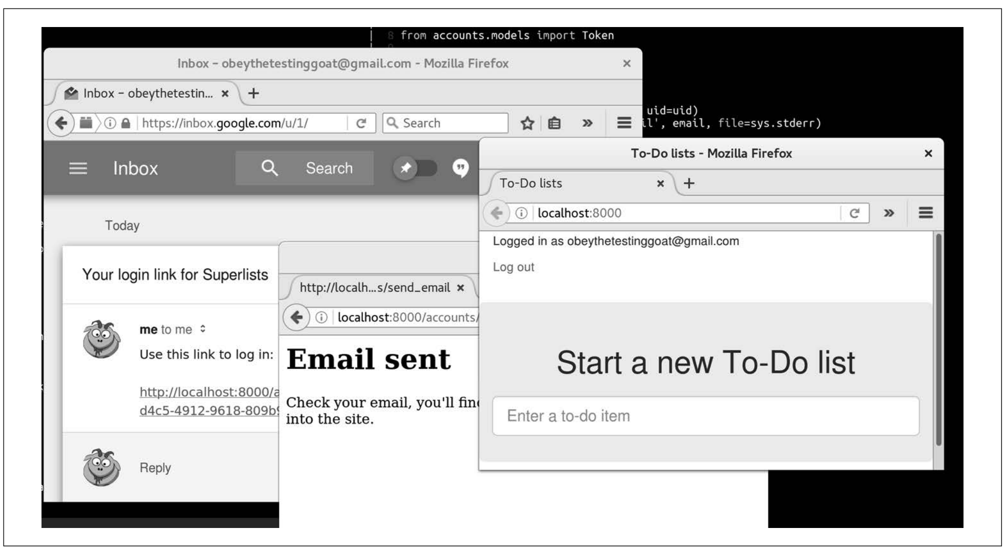


图 18-1：能正常使用！能正常使用！哇哈哈

> 如果遇到 `SMTPSenderRefused` 错误消息，可能是你忘了在运行开发服务器的 shell 中设定 `EMAIL_PASSWORD` 环境变量。

大概就是这样！这一过程看着简单，但我当时可是历经磨难。我在 Gmail 账户的安全界面四

处设置了好久，又在自定义用户模型的过程中少写了几个属性（因为我没认真阅读文档），甚至还以为自己发现了一个缺陷而换到 Django 开发版本，不过最终证明那并不是缺陷。

> **在 stderr 中记录错误**
>
> 探究时尤为重要的一点是能看到代码抛出的异常。Django 很讨厌，默认情况下并没把所有异常都输送到终端，不过可以在 settings.py 中使用 `LOGGING` 变量让 Django 这么做：
>
> *superlists/settings.py (ch16l017)*
>
> ```
> LOGGING = {
>     'version': 1,
>     'disable_existing_loggers': False,
>     'handlers': {
>         'console': {
>             'level': 'DEBUG',
>             'class': 'logging.StreamHandler',
>         },
>     },
>     'loggers': {
>         'django': {
>             'handlers': ['console'],
>         },
>     },
>     'root': {'level': 'INFO'},
> }
> ```
>
> Django 使用的是 Python 标准库中的企业级日志包。这个包虽然功能完善，但是学习曲线太陡。第 21 章将简单介绍一下这个包，Django 文档中有更详细的说明。

但不管怎么说，我们都实现了一个可用的方案！下面提交到 `passwordless-spike` 分支：

```
$ git status
$ git add accounts
$ git commit -am "spiked in custom passwordless auth backend"
```

接下来该去掉探究代码了。

18.3　去掉探究代码

去掉探究代码意味着要使用 TDD 重写原型代码。我们现在掌握了足够的信息，知道怎么做才是对的。那第一步做什么呢？当然是编写功能测试！

我们还得继续待在 `passwordless-spike` 分支中，看功能测试能否在探究代码中通过，然后再回到 `master` 分支，并且只提交功能测试。

下面是我们编写的第一版功能测试，很简单：

functional_tests/test_login.py

```
from django.core import mail
from selenium.webdriver.common.keys import Keys
import re

from .base import FunctionalTest

TEST_EMAIL = 'edith@example.com'
SUBJECT = 'Your login link for Superlists'


class LoginTest(FunctionalTest):

    def test_can_get_email_link_to_log_in(self):
        # 伊迪丝访问这个很棒的超级列表网站
        # 第一次注意到导航栏中有“登录”区域
        # 看到要求输入电子邮件地址，她便输入了
        self.browser.get(self.live_server_url)
        self.browser.find_element_by_name('email').send_keys(TEST_EMAIL)
        self.browser.find_element_by_name('email').send_keys(Keys.ENTER)

        # 出现一条消息，告诉她邮件已经发出
        self.wait_for(lambda: self.assertIn(
            'Check your email',
            self.browser.find_element_by_tag_name('body').text
        ))

        # 她查看邮件，看到一条消息
        email = mail.outbox[0]  ❶
        self.assertIn(TEST_EMAIL, email.to)
        self.assertEqual(email.subject, SUBJECT)

        # 邮件中有个URL链接
        self.assertIn('Use this link to log in', email.body)
        url_search = re.search(r'http://.+/.+$', email.body)
        if not url_search:
            self.fail(f'Could not find url in email body:\n{email.body}')
        url = url_search.group(0)
        self.assertIn(self.live_server_url, url)

        # 她点击了链接
        self.browser.get(url)

        # 她登录了！
        self.wait_for(
            lambda: self.browser.find_element_by_link_text('Log out')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn(TEST_EMAIL, navbar.text)
```

❶ 你是不是因不知如何在测试中获取邮件内容而担心？好消息是，我们暂时可以作弊！运行测试时，Django 允许通过 `mail.outbox` 属性访问服务器发送的电子邮件。稍后再说明如何检查“真实的”电子邮件（少安毋躁）。

运行这个功能测试，它能通过：

```
$ python manage.py test functional_tests.test_login
[...]
Not Found: /favicon.ico
saving uid [...]
login view
uid [...]
got token
new user

.
 ---------------------------------------------------------------------
Ran 1 test in 3.729s

OK
```

你甚至会看到我在探究视图的实现时留下的调试输出。现在该还原这些临时改动，然后再以测试驱动的方式重新逐一介绍了。

删除探究代码

```
$ git checkout master # 切换到master分支
$ rm -rf accounts # 删除所有探究代码
$ git add functional_tests/test_login.py
$ git commit -m "FT for login via email"
```

然后再次运行功能测试，让它驱动我们开发：

```
$ python manage.py test functional_tests.test_login
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [name="email"]
[...]
```

测试首先要求我们添加一个电子邮件地址输入框。Bootstrap 为导航栏提供了内置类，我们将使用它们。这个输入框放在一个表单里：

lists/templates/base.html (ch16l020)

```
<div class="container">

  <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <a class="navbar-brand" href="/">Superlists</a>
      <form class="navbar-form navbar-right" method="POST" action="#">
        <span>Enter email to log in:</span>
        <input class="form-control" name="email" type="text" />
        {% csrf_token %}
      </form>
    </div>
  </nav>

  <div class="row">
  [...]
```

现在功能测试是失败的，因为这个登录表单什么也没做：

```
$ python manage.py test functional_tests.test_login
[...]
AssertionError: 'Check your email' not found in 'Superlists\nEnter email to log
in:\nStart a new To-Do list'
```

我建议你现在像前面说过的那样设置 LOGGING。现在没必要特别测试这个，目前的测试组件会在出现异常时通知我们。到第 21 章你会发现，这个设置对调试非常有用。

该编写一些 Django 代码了。首先，创建一个名为 accounts 的应用，用于存放与登录有关的所有文件：

```
$ python manage.py startapp accounts
```

你现在还可以提交一次，把应用的占位文件与后面的修改区分开。

下面来重新构建这个极简的用户模型，不过这一次有测试。看看是否比探究时的更简洁。

18.4　一个极简的自定义用户模型

Django 内置的用户模型对记录用户信息做了各种设想，明确要记录的包括名和姓[3]，而且强制使用用户名。我坚信，除非真的需要，否则不要存储用户的任何信息。所以，一个只记录电子邮件地址的用户模型对我来说足够了。

我相信你已经知道我们应该创建 tests 文件夹，并在其中创建 __init__.py 文件，然后删除 tests.py，再新建 test_models.py，写入下述内容：

accounts/tests/test_models.py (ch16l024)

```
from django.test import TestCase
from django.contrib.auth import get_user_model

User = get_user_model()


class UserModelTest(TestCase):

    def test_user_is_valid_with_email_only(self):
        user = User(email='a@b.com')
        user.full_clean()  # 不该抛出异常
```

测试的结果是一个预期失败：

```
django.core.exceptions.ValidationError: {'password': ['This field cannot be
blank.'], 'username': ['This field cannot be blank.']}
```

密码？用户名？不！把模型写成这样如何？

注 3：虽然 Django 的重要维护者对这个决策并不后悔，但并非每个人都有名和姓。

accounts/models.py

```
from django.db import models

class User(models.Model):
    email = models.EmailField()
```

然后在 settings.py 中把 `accounts` 应用添加到 `INSTALLED_APPS` 中，再设定 `AUTH_USER_MODEL`：

superlists/settings.py (ch16l026)

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
    'accounts',
]

AUTH_USER_MODEL = 'accounts.User'
```

现在得到的错误与数据库有关：

```
django.db.utils.OperationalError: no such table: accounts_user
```

与之前一样，这表明我们要执行迁移。但当我们执行迁移时，Django 又会抱怨用户模型缺少一些元数据：

```
$ python manage.py makemigrations
Traceback (most recent call last):
[...]
    if not isinstance(cls.REQUIRED_FIELDS, (list, tuple)):
AttributeError: type object 'User' has no attribute 'REQUIRED_FIELDS'
```

唉，Django 啊，这个模型只有一个字段而已，你自己应该能找到问题的答案啊。既然你不能，那我就提供给你吧：

accounts/models.py

```
class User(models.Model):
    email = models.EmailField()
    REQUIRED_FIELDS = []
```

还有疑问吗？[4]

```
$ python manage.py makemigrations
[...]
AttributeError: type object 'User' has no attribute 'USERNAME_FIELD'
```

注 4：你可能想问，既然我觉得 Django 很笨，为什么不提交“合并请求”（pull request）修正呢？这个问题应该很容易修正。嗯，我保证等我写完这本书之后会这么做的。尖锐的批评就此打住吧！

经过几次失败之后，得到的模型如下所示：

accounts/models.py

```
class User(models.Model):
    email = models.EmailField()

    REQUIRED_FIELDS = []
    USERNAME_FIELD = 'email'
    is_anonymous = False
    is_authenticated = True
```

现在出现的错误稍有不同：

```
$ python manage.py makemigrations
SystemCheckError: System check identified some issues:

ERRORS:
accounts.User: (auth.E003) 'User.email' must be unique because it is named as
the 'USERNAME_FIELD'.
```

好吧，可以这样修正：

accounts/models.py (ch16l028-1)

```
    email = models.EmailField(unique=True)
```

现在能成功迁移了：

```
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model User
```

测试也能通过了：

```
$ python manage.py test accounts
[...]
Ran 1 tests in 0.001s
OK
```

不过用户模型比想象的复杂了一点，除了 `email` 字段之外，还有自动生成的 ID 字段，用作主键。这个模型可以进一步简化！

把测试当作文档

下面我们要把 `email` 字段设为主键，[5] 因而必须把自动生成的 `id` 字段删除。

我们可以直接这么做，测试也不会失败，然后信心满满地声称这“只是一次重构”。但是，最好为此专门编写一个测试：

注 5：其实电子邮件地址不太适合作为主键。一位深受其害的读者写了一封邮件给我，控诉了他们这十几年来使用电子邮件地址作为主键遇到的各种问题（因为这样无法管理多用户账户）。所以，还是那句话，具体问题具体分析。

accounts/tests/test_models.py (ch16l028-3)

```
    def test_email_is_primary_key(self):
        user = User(email='a@b.com')
        self.assertEqual(user.pk, 'a@b.com')
```

如果以后回过头再看代码，这个测试能唤起我们的记忆，想起曾经做过这次修改。

```
    self.assertEqual(user.pk, 'a@b.com')
AssertionError: None != 'a@b.com'
```

> 测试可以作为一种文档形式，因为测试体现了你对某个类或函数的需求。如果你忘记了为什么要用某种方法编写代码，可以回过头来看测试，有时就能找到答案。这就是一定要给测试方法起个意思明确的名字的原因。

实现的方式如下（可以先使用 `unique=True`，看看结果如何）：

accounts/models.py (ch16l028-4)

```
    email = models.EmailField(primary_key=True)
```

一定不能忘了调整迁移：

```
$ rm accounts/migrations/0001_initial.py
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model User
```

现在两个测试都能通过：

```
$ python manage.py test accounts
[...]
Ran 2 tests in 0.001s
OK
```

18.5　令牌模型：把电子邮件地址与唯一的ID关联起来

接下来构建一个令牌模型。下面是个简短的单元测试，能捕获基本的问题——应该能把电子邮件地址关联到唯一的 ID 上，而且 ID 不能在一行中重复出现：

accounts/tests/test_models.py (ch16l030)

```
from accounts.models import Token
[...]


class TokenModelTest(TestCase):
```

```
    def test_links_user_with_auto_generated_uid(self):
        token1 = Token.objects.create(email='a@b.com')
        token2 = Token.objects.create(email='a@b.com')
        self.assertNotEqual(token1.uid, token2.uid)
```

使用 TDD 驱动开发 Django 模型要历经几番波折，因为这涉及迁移，所以我们将像这样迭代多次：微改代码、创建迁移、遇到新错误、删除迁移、重新创建迁移、再修改代码……

```
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0002_token.py
    - Create model Token
$ python manage.py test accounts
[...]
TypeError: 'email' is an invalid keyword argument for this function
```

我相信你能按部就班走完整个过程。记住，虽然我看不见你，但是测试山羊能！

```
$ rm accounts/migrations/0002_token.py
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0002_token.py
    - Create model Token
$ python manage.py test accounts
AttributeError: 'Token' object has no attribute 'uid'
```

最终写出的模型代码如下：

accounts/models.py (ch16l033)

```
class Token(models.Model):
    email = models.EmailField()
    uid = models.CharField(max_length=40)
```

得到的错误是：

```
$ python manage.py test accounts
[...]

    self.assertNotEqual(token1.uid, token2.uid)
AssertionError: '' == ''
```

现在要决定如何生成随机的唯一 ID 字段。我们可以使用 `random` 模块，但是 Python 自带的另一个模块是专门用于生成唯一 ID 的，名为“uuid”（universally unique id 的简称）。

它的用法如下：

accounts/models.py (ch16l035)

```
import uuid
[...]

class Token(models.Model):
    email = models.EmailField()
    uid = models.CharField(default=uuid.uuid4, max_length=40)
```

再调整一下迁移，测试便能通过：

```
$ python manage.py test accounts
[...]
Ran 3 tests in 0.015s

OK
```

不错，逐渐走上正轨了——至少模型层完成了。下一章将介绍模拟技术，这是测试外部依赖（例如邮件）的关键技术。

探索性编程、探究及去掉探究代码

- 探究
  为了学习新 API 或调查新方案的可行性而做的探索性编程。没有测试也能探究。最好在一个新分支中探究，去掉探究代码时再回到主分支。
- 去掉探究代码
  把探究所得应用到真实的代码基中。要完全摒弃探究代码，然后从头开始，用 TDD 流程再实现一次。去掉探究代码后实际编写的代码往往与最初有很大不同，而且通常更好。
- 针对探究代码编写功能测试
  该不该这么做要视情况而定。支持这么做的人认为，这样有助于正确编写功能测试——找出测试探究的方法与探究本身一样具有挑战性；不支持这么做的人觉得这样会阻碍思路，写出的代码往往与探究时很像——我们要力求避免这种情况。

第 19 章

使用驭件测试外部依赖或减少重复

本章开始说明如何测试发送电子邮件的代码。通过前面的功能测试，我们得知 Django 提供了获取所发送邮件的方式，即 `mail.outbox` 属性。不过我想在这一章演示一种十分重要的测试技术，叫作**模拟技术**。鉴于此，本章的单元测试将假装 Django 没有提供这样的便捷方式。

> 我是说不能使用 Django 的 `mail.outbox` 吗？不是。你应该使用它，因为它很便利。但是我想教你模拟技术，因为这是在单元测试中测试外部依赖的通用方式。毕竟你不会一直使用 Django。即便如此，除了发送电子邮件之外还有很多操作，只要与第三方 API 交互都适合使用驭件测试。

19.1 开始之前布好基本管道

首先设置一个基本的视图和 URL。编写一个简单的测试，检查发送登录邮件的 URL 最终会重定向到首页：

accounts/tests/test_views.py

```
from django.test import TestCase


class SendLoginEmailViewTest(TestCase):

    def test_redirects_to_home_page(self):
        response = self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })
        self.assertRedirects(response, '/')
```

我们已经在 accounts/urls.py 中设置了 `url`，也在 superlists/urls.py 中使用 `include` 引入了，再加上下述代码，这个测试便能通过：

accounts/views.py

```
from django.core.mail import send_mail
from django.shortcuts import redirect

def send_login_email(request):
    return redirect('/')
```

现在导入 `send_mail` 函数只是占个位子：

```
$ python manage.py test accounts
[...]
Ran 4 tests in 0.015s

OK
```

好的，有切入点了，下面开始使用模拟技术。

19.2 自己动手模拟（打猴子补丁）

在真实情况中，我们调用 `send_mail` 时希望 Django 连接电子邮件服务提供商，通过网络把真实的电子邮件发送出去。但这不是我们希望在测试中发生的。代码有外部副作用时也是如此，例如调用 API、发推文、发短信，等等。在单元测试中，我们并不想真的通过互联网发推文或调用 API。可是，我们必须找到一种方法，测试代码是否正确。驭件[1]正是我们寻找的答案。

恰好，Python 的优势之一是它的动态本性，这样十分有利于模拟，我们有时称这样的做法为打猴子补丁。首先，假设调用 `send_mail` 时要设定邮件主题、发件地址和收件地址，比如像下面这样：

accounts/views.py

```
def send_login_email(request):
    email = request.POST['email']
    # send_mail(
    #     'Your login link for Superlists',
    #     'body text tbc',
    #     'noreply@superlists',
    #     [email],
    # )
    return redirect('/')
```

在不**真正**调用 `send_mail` 函数的情况下，应该怎么测试呢？答案是在调用 `send_login_email`

注 1：我使用的是通用术语“驭件”（mock），而测试狂热者却希望明确区分“测试替身”这类测试工具中的不同概念，包括侦件（spy）、伪件（fake）和桩件（stub）。在这本书中不必纠结它们之间的区别，如果你想深入了解，请阅读 Justin Searls 写的精彩维基“Test Double”。剧透：此文充满各种测试知识。

视图之前，测试可以让 Python 在运行时把 `send_mail` 函数替换成一个伪造的版本。看一下具体的代码：

accounts/tests/test_views.py (ch17l005)

```
from django.test import TestCase
import accounts.views   ❷

class SendLoginEmailViewTest(TestCase):
    [...]

    def test_sends_mail_to_address_from_post(self):
        self.send_mail_called = False

        def fake_send_mail(subject, body, from_email, to_list):   ❶
            self.send_mail_called = True
            self.subject = subject
            self.body = body
            self.from_email = from_email
            self.to_list = to_list

        accounts.views.send_mail = fake_send_mail   ❷

        self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })


        self.assertTrue(self.send_mail_called)
        self.assertEqual(self.subject, 'Your login link for Superlists')
        self.assertEqual(self.from_email, 'noreply@superlists')
        self.assertEqual(self.to_list, ['edith@example.com'])
```

❶ 定义 `fake_send_mail` 函数。它看起来与 `send_mail` 函数很像，但它其实只是使用 `self` 的一些变量存储了一些关于调用方式的信息。

❷ 然后，在测试执行 `self.client.post` 之前，把真的 `accounts.views.send_mail` 函数换成假的——只需一次简单的赋值。

注意，我们没有施什么魔法，只是打破常规，利用 Python 这门动态语言的优势。

在真正调用函数之前，只要命名空间正确，就可以修改用于访问函数的变量（因此才需要导入位于顶层的 `accounts` 模块，这样方能进入 `accounts.views` 模块，达到 `accounts.views.send_login_email` 函数所在的作用域）。

这种做法不只限于单元测试，在任何 Python 代码中都可以像这样打猴子补丁。

你可能要花点时间才能习惯。在深入探讨细节之前，你要说服自己相信这没什么大不了的。

- 为什么要使用 `self` 传递信息？这只是在 `fake_send_mail` 函数的作用域内外传递信息的便利方式。此外，我们还可以使用可变的对象，例如列表或字典，只要那个对象在伪造函数的外部即可。（如果好奇，可以试试不同的方式，看哪些可行，哪些不可行。）

- 一定要在调用真实函数“之前”！我曾无数次呆坐在那里，百思不得其解，为什么驭件不起作用呢？最后才恍然大悟，我没有在调用真实的函数**之前**替换为伪造的函数。

下面看一下我们自己动手打造的驭件能否驱动开发：

```
$ python manage.py test accounts
[...]
    self.assertTrue(self.send_mail_called)
AssertionError: False is not true
```

直接调用 `send_mail` 试试：

accounts/views.py

```
def send_login_email(request):
    send_mail()
    return redirect('/')
```

测试结果变成了：

```
TypeError: fake_send_mail() missing 4 required positional arguments: 'subject',
'body', 'from_email', and 'to_list'
```

看样子猴子补丁起作用了！我们调用了 `send_mail`，而它执行的是 `fake_send_mail` 函数，后者要求提供更多参数。提供参数试试：

accounts/views.py

```
def send_login_email(request):
    send_mail('subject', 'body', 'from_email', ['to email'])
    return redirect('/')
```

测试的结果为：

```
    self.assertEqual(self.subject, 'Your login link for Superlists')
AssertionError: 'subject' != 'Your login link for Superlists'
```

一切顺利。然后调整代码，改成下面这样：

accounts/views.py

```
def send_login_email(request):
    email = request.POST['email']
    send_mail(
        'Your login link for Superlists',
        'body text tbc',
        'noreply@superlists',
        [email]
    )
    return redirect('/')
```

测试能通过了：

```
$ python manage.py test accounts

Ran 5 tests in 0.016s

OK
```

棒极了！我们模拟了 `send_email` 函数，为正常情况下应该通过互联网发送邮件的代码编写了测试，这样测试和代码就没有出入了。[2]

19.3 Python的模拟库

流行的 mock 包从 Python 3.3 起纳入了标准库。[3] 这个包提供了一个充满魔力的对象，名为 `mock`。下面在 Python shell 中试用一下：

```
>>> from unittest.mock import Mock
>>> m = Mock()
>>> m.any_attribute
<Mock name='mock.any_attribute' id='140716305179152'>
>>> type(m.any_attribute)
<class 'unittest.mock.Mock'>
>>> m.any_method()
<Mock name='mock.any_method()' id='140716331211856'>
>>> m.foo()
<Mock name='mock.foo()' id='140716331251600'>
>>> m.called
False
>>> m.foo.called
True
>>> m.bar.return_value = 1
>>> m.bar(42, var='thing')
1
>>> m.bar.call_args
call(42, var='thing')
```

这个对象很神奇，它能响应任何属性访问或方法调用，可以指明调用的返回值，还可以审查调用时传入的参数是什么。看起来它很适合在单元测试中使用。

19.3.1 使用unittest.patch

如果觉得这不够用，`mock` 模块还提供了辅助函数 `patch`，利用它可以实现前面动手打的猴子补丁。

稍后再讲原理，现在先看具体用法：

accounts/tests/test_views.py (ch17l007)

```
from django.test import TestCase
from unittest.mock import patch
[...]

    @patch('accounts.views.send_mail')
    def test_sends_mail_to_address_from_post(self, mock_send_mail):
        self.client.post('/accounts/send_login_email', data={
```

注 2：是的，我知道 Django 已经使用 `mail.outbox` 提供了电子邮件驭件。但是，再次声明，我们要假装没有。如果我们使用的是 Flask 呢？或者，如果这是调用 API，而不是发送邮件呢？

注 3：Python 2 用户可以使用 `pip install mock` 安装。

```
            'email': 'edith@example.com'
        })

        self.assertEqual(mock_send_mail.called, True)
        (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args
        self.assertEqual(subject, 'Your login link for Superlists')
        self.assertEqual(from_email, 'noreply@superlists')
        self.assertEqual(to_list, ['edith@example.com'])
```

重新运行这个测试，你会发现它仍能通过。大改之后依然能通过的测试最让人怀疑，下面故意搞个破坏：

accounts/tests/test_views.py (ch17l008)

```
        self.assertEqual(to_list, ['schmedith@example.com'])
```

并在视图中添加一行代码，打印调试信息：

accounts/views.py (ch17l009)

```
def send_login_email(request):
    email = request.POST['email']
    print(type(send_mail))
    send_mail(
        [...]
```

再次运行测试：

```
$ python manage.py test accounts
[...]
<class 'function'>
<class 'unittest.mock.MagicMock'>
[...]
AssertionError: Lists differ: ['edith@example.com'] !=
['schmedith@example.com']
[...]

Ran 5 tests in 0.024s

FAILED (failures=1)
```

显然，测试失败了。从失败消息前面的输出可以看出，`send_mail` 函数的类型在第一个单元测试中是常规的函数，而在第二个单元测试中则是一个驭件。

删掉故意出错的代码，然后深入分析到底发生了什么：

accounts/tests/test_views.py (ch17l011)

```
@patch('accounts.views.send_mail')  ❶
def test_sends_mail_to_address_from_post(self, mock_send_mail):  ❷
    self.client.post('/accounts/send_login_email', data={
        'email': 'edith@example.com'  ❸
    })

    self.assertEqual(mock_send_mail.called, True)  ❹
    (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args  ❺
```

```
        self.assertEqual(subject, 'Your login link for Superlists')
        self.assertEqual(from_email, 'noreply@superlists')
        self.assertEqual(to_list, ['edith@example.com'])
```

❶ patch 装饰器的参数是要打猴子补丁的函数的点分名称。这一行代码的作用等同于动手替换 accounts.views 中的 send_mail 函数。这个装饰器的优点很多，不仅可以自动把目标替换成驭件，结束后还能自动换回原对象（否则后续使用的仍是打过猴子补丁的版本，这可能对其他测试产生影响）。

❷ patch 通过传给测试方法的参数注入驭件。这个参数的名称随意，不过我习惯在原对象名称前面加上 mock_。

❸ 像往常一样调用要测试的函数，但是在测试方法中使用的是驭件，所以视图不会真的调用 send_mail，而是使用 mock_send_mail。

❹ 下断言，检查驭件在测试过程中有什么变化。先调用驭件……

❺ ……然后拆解出各个位置参数和关键字参数，检查调用时传入的是什么值。（稍后会详细讨论 call_args。）

彻底弄明白了吗？没有？没关系。后面还会在多个测试中使用驭件，你会逐渐习惯的。

19.3.2　让测试向前迈一小步

暂且回到功能测试，看看是在哪里失败的：

```
$ python manage.py test functional_tests.test_login
[...]
AssertionError: 'Check your email' not found in 'Superlists\nEnter email to log
in:\nStart a new To-Do list'
```

提交电子邮件地址目前没有任何效果，因为表单没向任何地方发送数据。下面在 base.html 中设置：[4]

lists/templates/base.html (ch17l012)

```
<form class="navbar-form navbar-right"
      method="POST"
      action="{% url 'send_login_email' %}">
```

有没有用？没有，还有错误。为什么呢？因为成功给用户发送电子邮件后没有显示成功消息。下面为此添加一个测试。

19.3.3　测试Django消息框架

我们将使用 Django 的“消息框架”，通常用于显示临时的“成功”或“提醒”消息，指明操作的结果。如果你没用过这个框架，可以看一下它的文档。

注 4：限于本书纸张的尺寸，我把这个 form 标签分成了三行。如果你没见过这种写法，可能觉得有点奇怪，但这是有效的 HTML。如果不喜欢，你可以不这样写。（☺）

测试 Django 的消息有点曲折，要把 `follow=True` 传给测试客户端，让它获取 302 重定向后的页面，在里面查找消息列表（在显示之前转换为列表）。这个测试如下所示：

accounts/tests/test_views.py (ch17l013)

```
    def test_adds_success_message(self):
        response = self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        }, follow=True)

        message = list(response.context['messages'])[0]
        self.assertEqual(
            message.message,
            "Check your email, we've sent you a link you can use to log in."
        )
        self.assertEqual(message.tags, "success")
```

测试的结果为：

```
$ python manage.py test accounts
[...]
    message = list(response.context['messages'])[0]
IndexError: list index out of range
```

然后像下面这样修改，让测试通过：

accounts/views.py (ch17l014)

```
from django.contrib import messages
[...]

def send_login_email(request):
    [...]
    messages.success(
        request,
        "Check your email, we've sent you a link you can use to log in."
    )
    return redirect('/')
```

驭件可能导致与实现紧密耦合

这个框注涉及中级测试技巧。如果第一次没读懂，读完本章和第 23 章之后再回过头来看。

我说过，测试消息有点曲折，我试了好几次才做对。其实，我们已经不在工作中这样测试消息了，而是使用驭件。使用驭件的话，上述测试可以改成这样：

accounts/tests/test_views.py (ch17l014-2)

```
from unittest.mock import patch, call
[...]

    @patch('accounts.views.messages')
    def test_adds_success_message_with_mocks(self, mock_messages):
        response = self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })

        expected = "Check your email, we've sent you a link you can use to log in."
        self.assertEqual(
            mock_messages.success.call_args,
            call(response.wsgi_request, expected),
        )
```

我们模拟了 messages 模块，然后检查调用 messages.success 时传入了正确的参数：原请求和想看到的消息。

使用前面的代码就能让这个测试通过。不过有个问题：对 messages 框架来说，获得相同结果的方式不止一种。视图的代码还可以像这样写：

accounts/views.py (ch17l014-3)

```
    messages.add_message(
        request,
        messages.SUCCESS,
        "Check your email, we've sent you a link you can use to log in."
    )
```

此时，未使用驭件的测试仍能通过，但使用驭件的测试将失败。这是因为没有调用 messages.success，而是调用了 messages.add_message。即便最终结果一样，而且代码也是“正确的”，可测试却出问题了。

这就是人们常说的，使用驭件可能导致“与实现紧密耦合”。我们知道，通常最好测试行为，而不测试实现细节；测试发生了什么，而不测试是如何发生的。驭件往往在“如何做”这条道上走得太远，而很少关注“是什么”。

后续章节还会深入讨论驭件的优缺点。

19.3.4 在HTML中添加消息

功能测试有进展吗？啊，还没有。我们要把消息添加到页面中，像下面这样：

lists/templates/base.html (ch17l015)

```
[...]
</nav>

{% if messages %}
  <div class="row">
```

```
          <div class="col-md-8">
            {% for message in messages %}
              {% if message.level_tag == 'success' %}
                <div class="alert alert-success">{{ message }}</div>
              {% else %}
                <div class="alert alert-warning">{{ message }}</div>
              {% endif %}
            {% endfor %}
          </div>
        </div>
      {% endif %}
```

现在该有进展了吧？是的！

```
$ python manage.py test accounts
[...]
Ran 6 tests in 0.023s

OK

$ python manage.py test functional_tests.test_login
[...]
AssertionError: 'Use this link to log in' not found in 'body text tbc'
```

失败消息提醒我们，在电子邮件的正文中找出用于点击登录的链接。

先暂时作弊，直接修改视图中的值：

accounts/views.py

```
        send_mail(
            'Your login link for Superlists',
            'Use this link to log in',
            'noreply@superlists',
            [email]
        )
```

这样，功能测试稍微向前走了一点：

```
$ python manage.py test functional_tests.test_login
[...]
AssertionError: Could not find url in email body:
Use this link to log in
```

19.3.5　构建登录URL

接下来该构建某种形式的 URL 了！这一次依然作弊：

accounts/tests/test_views.py (ch17l017)

```
class LoginViewTest(TestCase):

    def test_redirects_to_home_page(self):
        response = self.client.get('/accounts/login?token=abcd123')
        self.assertRedirects(response, '/')
```

假设令牌通过 GET 参数传递，即放在 ? 后面。现在还不需要它做些什么。

我相信你能构建出所需的 URL 和视图，在这个过程中你会历经下述错误。

- 没有 URL：

```
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

- 没有视图：

```
AttributeError: module 'accounts.views' has no attribute 'login'
```

- 视图出错：

```
ValueError: The view accounts.views.login didn't return an HttpResponse object.
It returned None instead.
```

- 测试通过：

```
$ python manage.py test accounts
[...]

Ran 7 tests in 0.029s

OK
```

现在，链接的目标 URL 有了，但是还没什么用，因为我们还没给用户提供令牌。

19.3.6　确认给用户发送了带有令牌的链接

对 send_login_email 视图来说，我们测试了电子邮件的主题、发件地址和收件地址，而包含令牌或 URL 的正文还没有测试。下面为正文编写两个测试：

accounts/tests/test_views.py (ch17l021)

```
from accounts.models import Token
[...]

    def test_creates_token_associated_with_email(self):
        self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })
        token = Token.objects.first()
        self.assertEqual(token.email, 'edith@example.com')


    @patch('accounts.views.send_mail')
    def test_sends_link_to_login_using_token_uid(self, mock_send_mail):
        self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })

        token = Token.objects.first()
        expected_url = f'http://testserver/accounts/login?token={token.uid}'
        (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args
        self.assertIn(expected_url, body)
```

第一个测试十分简单，检查我们在数据库中创建的令牌是与 POST 请求中的电子邮件地址关联的。

第二个测试是我们第二次使用驭件。再次使用 `patch` 装饰器模拟 `send_mail` 函数，但这次关注的是调用时传入的 `body` 参数。

现在，这些测试是失败的，因为还没创建令牌：

```
$ python manage.py test accounts
[...]
AttributeError: 'NoneType' object has no attribute 'email'
[...]
AttributeError: 'NoneType' object has no attribute 'uid'
```

创建令牌就能让第一个测试通过：

accounts/views.py (ch17l022)

```
from accounts.models import Token
[...]

def send_login_email(request):
    email = request.POST['email']
    token = Token.objects.create(email=email)
    send_mail(
        [...]
```

第二个测试提示我们在电子邮件的正文中使用令牌：

```
[...]
AssertionError:
'http://testserver/accounts/login?token=[...]
not found in 'Use this link to log in'

FAILED (failures=1)
```

因此，像下面这样在电子邮件中插入令牌：

accounts/views.py (ch17l023)

```
from django.core.urlresolvers import reverse
[...]

def send_login_email(request):
    email = request.POST['email']
    token = Token.objects.create(email=email)
    url = request.build_absolute_uri(    ❶
        reverse('login') + '?token=' + str(token.uid)
    )
    message_body = f'Use this link to log in:\n\n{url}'
    send_mail(
        'Your login link for Superlists',
        message_body,
        'noreply@superlists',
        [email]
    )
[...]
```

❶ 请注意 `request.build_absolute_uri`，这是在 Django 中构建“完整”URL 的一种方式，所得的 URL 包括域名和协议（http/https）。除此之外还有其他方式，不过往往都牵扯“网站”框架，容易导致代码过于复杂。如果你好奇，简单搜索一下就能找到很多这方面的讨论。

我们又解决了两个问题。接下来，需要一个身份验证后端，检查令牌的有效性，然后返回对应的用户。此外，还需要让登录视图在用户通过身份验证后登入用户。

19.4 去除自定义的身份验证后端中的探究代码

接下来要自定义身份验证后端。探究时编写的代码如下所示：

```
class PasswordlessAuthenticationBackend(object):

    def authenticate(self, uid):
        print('uid', uid, file=sys.stderr)
        if not Token.objects.filter(uid=uid).exists():
            print('no token found', file=sys.stderr)
            return None
        token = Token.objects.get(uid=uid)
        print('got token', file=sys.stderr)
        try:
            user = ListUser.objects.get(email=token.email)
            print('got user', file=sys.stderr)
            return user
        except ListUser.DoesNotExist:
            print('new user', file=sys.stderr)
            return ListUser.objects.create(email=token.email)


    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

这段代码的意思是：

- 检查数据库中是否有指定的 UID；
- 如果没有，返回 `None`；
- 如果有，提取电子邮件地址，通过这个地址找到现有的用户，或者创建一个新用户。

19.4.1 一个`if`语句需要一个测试

如何为这种函数编写测试有个经验法则：一个 `if` 语句需要一个测试，一个 `try/except` 语句也需要一个测试。所以这里一共需要三个测试。像下面这样写怎么样？

accounts/tests/test_authentication.py

```
from django.test import TestCase
from django.contrib.auth import get_user_model
from accounts.authentication import PasswordlessAuthenticationBackend
from accounts.models import Token
User = get_user_model()
```

```
class AuthenticateTest(TestCase):

    def test_returns_None_if_no_such_token(self):
        result = PasswordlessAuthenticationBackend().authenticate(
            'no-such-token'
        )
        self.assertIsNone(result)


    def test_returns_new_user_with_correct_email_if_token_exists(self):
        email = 'edith@example.com'
        token = Token.objects.create(email=email)
        user = PasswordlessAuthenticationBackend().authenticate(token.uid)
        new_user = User.objects.get(email=email)
        self.assertEqual(user, new_user)


    def test_returns_existing_user_with_correct_email_if_token_exists(self):
        email = 'edith@example.com'
        existing_user = User.objects.create(email=email)
        token = Token.objects.create(email=email)
        user = PasswordlessAuthenticationBackend().authenticate(token.uid)
        self.assertEqual(user, existing_user)
```

在 authenticate.py 中先放一个占位函数：

accounts/authentication.py

```
class PasswordlessAuthenticationBackend(object):

    def authenticate(self, uid):
        pass
```

测试的结果如何？

```
$ python manage.py test accounts

.FE.........
======================================================================
ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/accounts/tests/test_authentication.py", line 21, in
test_returns_new_user_with_correct_email_if_token_exists
    new_user = User.objects.get(email=email)
[...]
accounts.models.DoesNotExist: User matching query does not exist.

======================================================================
FAIL: test_returns_existing_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
 ---------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "/.../superlists/accounts/tests/test_authentication.py", line 30, in
test_returns_existing_user_with_correct_email_if_token_exists
    self.assertEqual(user, existing_user)
AssertionError: None != <User: User object>

 ---------------------------------------------------------------------
Ran 12 tests in 0.038s

FAILED (failures=1, errors=1)
```

修改代码试试：

accounts/authentication.py (ch17l026)

```
from accounts.models import User, Token

class PasswordlessAuthenticationBackend(object):

    def authenticate(self, uid):
        token = Token.objects.get(uid=uid)
        return User.objects.get(email=token.email)
```

一个测试通过了，但却导致另一个失败了：

```
$ python manage.py test accounts
ERROR: test_returns_None_if_no_such_token
(accounts.tests.test_authentication.AuthenticateTest)

accounts.models.DoesNotExist: Token matching query does not exist.

ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
[...]
accounts.models.DoesNotExist: User matching query does not exist.
```

下面逐个修正：

accounts/authentication.py (ch17l027)

```
    def authenticate(self, uid):
        try:
            token = Token.objects.get(uid=uid)
            return User.objects.get(email=token.email)
        except Token.DoesNotExist:
            return None
```

只剩一个失败测试了：

```
ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
[...]
accounts.models.DoesNotExist: User matching query does not exist.

FAILED (errors=1)
```

这个问题可以像这样解决：

accounts/authentication.py (ch17l028)

```
    def authenticate(self, uid):
        try:
            token = Token.objects.get(uid=uid)
            return User.objects.get(email=token.email)
        except User.DoesNotExist:
            return User.objects.create(email=token.email)
        except Token.DoesNotExist:
            return None
```

这比探究时编写的代码更简洁！

19.4.2　get_user方法

我们已经实现了供 Django 用于登入新用户的 `authenticate` 函数。这个协议的第二部分是实现 `get_user` 方法，它的作用是根据唯一标识符（电子邮件地址）获取用户，如果找不到就返回 `None`（如果你记不清了，请看一下本节开头给出的探究代码）。

下面为这两个需求编写几个测试：

accounts/tests/test_authentication.py (ch17l030)

```
class GetUserTest(TestCase):

    def test_gets_user_by_email(self):
        User.objects.create(email='another@example.com')
        desired_user = User.objects.create(email='edith@example.com')
        found_user = PasswordlessAuthenticationBackend().get_user(
            'edith@example.com'
        )
        self.assertEqual(found_user, desired_user)


    def test_returns_None_if_no_user_with_that_email(self):
        self.assertIsNone(
            PasswordlessAuthenticationBackend().get_user('edith@example.com')
        )
```

这时的失败消息是：

```
AttributeError: 'PasswordlessAuthenticationBackend' object has no attribute
'get_user'
```

那就定义一个占位方法：

accounts/authentication.py (ch17l031)

```
class PasswordlessAuthenticationBackend(object):

    def authenticate(self, uid):
        [...]
```

```
    def get_user(self, email):
        pass
```

现在失败消息变成了：

```
    self.assertEqual(found_user, desired_user)
AssertionError: None != <User: User object>
```

慢慢实现这个方法（一步一步来，看测试是否像我们设想的那样失败）：

accounts/authentication.py (ch17l033)

```
    def get_user(self, email):
        return User.objects.first()
```

现在第一个断言通过了，失败消息变成了：

```
    self.assertEqual(found_user, desired_user)
AssertionError: <User: User object> != <User: User object>
```

那就调用 `get`，并传入电子邮件地址：

accounts/authentication.py (ch17l034)

```
    def get_user(self, email):
        return User.objects.get(email=email)
```

现在针对返回 `None` 的测试失败了：

```
ERROR: test_returns_None_if_no_user_with_that_email
[...]
accounts.models.DoesNotExist: User matching query does not exist.
```

根据提示，可以这样完成整个方法：

accounts/authentication.py (ch17l035)

```
    def get_user(self, email):
        try:
            return User.objects.get(email=email)
        except User.DoesNotExist:
            return None  ❶
```

❶ 这里可以使用 `pass`，函数默认会返回 `None`。但我们希望它明确返回 `None`，所以根据“明了胜于晦涩”原则，应该返回 `None`。

现在测试能通过了：

```
OK
```

至此，我们得到了一个可用的身份验证后端。

19.4.3 在登录视图中使用自定义的验证后端

最后一步，在登录视图中使用这个后端。首先，在 settings.py 中添加自定义的后端：

superlists/settings.py (ch17l036)

```
AUTH_USER_MODEL = 'accounts.User'
AUTHENTICATION_BACKENDS = [
    'accounts.authentication.PasswordlessAuthenticationBackend',
]

[...]
```

然后编写几个测试，检查视图的行为。再看一下探究时编写的视图：

accounts/views.py

```
def login(request):
    print('login view', file=sys.stderr)
    uid = request.GET.get('uid')
    user = auth.authenticate(uid=uid)
    if user is not None:
        auth.login(request, user)
    return redirect('/')
```

视图要调用 `django.contrib.auth.authenticate`。如果返回一个用户，再调用 `django.contrib.auth.login`。

> 现在应该阅读 Django 文档中对身份验证的说明，进一步了解细节。

19.5　使用驭件的另一个原因：减少重复

我们已经使用驭件测试了外部依赖，例如 Django 发送电子邮件的功能。使用驭件的主要原因是隔离外部副作用，这里是想避免在测试中真的发送电子邮件。

本节将说明驭件的另一种用法。此时，没有什么副作用需要担心，但是鉴于一些原因，我们仍然想使用驭件。

如果不使用驭件，测试登录视图要检查用户有没有真的登入，即检查在正确的情况下有没有为用户赋予表明已通过身份验证的会话 cookie。

但是，我们自定义的身份验证后端有几个不同的代码执行路径：令牌无效时返回 `None`、用户存在时返回既存用户、令牌有效但用户不存在时新建用户。因此，为了全面测试这个视图，要分别针对这三种情况编写测试。

> 如果能有效减少测试之间的重复，就有充分的理由使用驭件。这是避免**组合爆炸**的一种方式。

此外，我们使用的是 Django 的 `auth.authenticate` 函数，而没有直接调用自己的代码，这样便于以后添加额外的后端。

因此，有必要考虑这里的实现细节（与“驭件可能导致与实现紧密耦合”框注中所述的相反），而使用驭件能避免在多个测试中重复编写实现方式。下面看一下这个测试应该怎样使用驭件编写：

accounts/tests/test_views.py (ch17l037)

```
from unittest.mock import patch, call
[...]

    @patch('accounts.views.auth')   ❶
    def test_calls_authenticate_with_uid_from_get_request(self, mock_auth):   ❷
        self.client.get('/accounts/login?token=abcd123')
        self.assertEqual(
            mock_auth.authenticate.call_args,   ❸
            call(uid='abcd123')   ❹
        )
```

❶ 我们期望在 views.py 中使用 `django.contrib.auth` 模块，所以这里模拟它的行为。注意，这里模拟的不是一个函数，而是整个模块，即模拟模块中的全部函数（及其他对象）。

❷ 与之前一样，把被模拟的对象注入测试方法。

❸ 这一次模拟的是一个模块，而不是一个函数。所以，`call_args` 不能在 `mock_auth` 模块上检查，而要在 `mock_auth.authenticate` 函数上检查。因为驭件的所有属性都是驭件，所以 `mock_auth.authenticate` 函数也是一个驭件。由此可以看出，与自己动手相比，mock 对象是多么好用。

❹ 这一次没有“拆包”调用参数，而是使用更简洁的 `call` 函数，指明调用时应该传入什么参数——GET 请求中的令牌。（参见下述框注。）

关于驭件的 call_args

驭件的 `call_args` 属性表示调用驭件时传入的位置参数和关键字参数。这是一个特殊的“调用”对象类型，其实是一个元组，内容为 `(positional_args, keyword_args)`。其中 `positional_args` 自身也是一个元组，包含各个位置参数；而 `keyword_args` 是个字典。

```
>>> from unittest.mock import Mock, call
>>> m = Mock()
>>> m(42, 43, 'positional arg 3', key='val', thing=666)
<Mock name='mock()' id='139909729163528'>

>>> m.call_args
call(42, 43, 'positional arg 3', key='val', thing=666)

>>> m.call_args == ((42, 43, 'positional arg 3'), {'key': 'val', 'thing': 666})
True
>>> m.call_args == call(42, 43, 'positional arg 3', key='val', thing=666)
True
```

所以，在上述测试中还可以这么写：

accounts/tests/test_views.py

```
        self.assertEqual(
            mock_auth.authenticate.call_args,
            ((,), {'uid': 'abcd123'})
        )
        # 或者这样写
        args, kwargs = mock_auth.authenticate.call_args
        self.assertEqual(args, (,))
        self.assertEqual(kwargs, {'uid': 'abcd123')
```

不过可以看出，使用 call 辅助函数更简洁。

测试是什么情况呢？第一个错误是：

```
$ python manage.py test accounts
[...]
AttributeError: <module 'accounts.views' from
'/.../superlists/accounts/views.py'> does not have the attribute 'auth'
```

在使用驭件的测试中，第一个失败消息经常是 module foo does not have the attribute bar。这个消息的意思是，你尝试模拟的东西在目标模块中还不存在（或者尚未导入）。

导入 django.contrib.auth 后，错误会变：

accounts/views.py (ch17l038)

```
from django.contrib import auth, messages
[...]
```

现在的错误是：

```
AssertionError: None != call(uid='abcd123')
```

测试指出，视图根本没有调用 auth.authenticate 函数。下面修正，但是故意做错，看看效果如何：

accounts/views.py (ch17l039)

```
def login(request):
    auth.authenticate('bang!')
    return redirect('/')
```

调用的确实是 bang!：

```
$ python manage.py test accounts
[...]
AssertionError: call('bang!') != call(uid='abcd123')
[...]
FAILED (failures=1)
```

下面给 authenticate 提供预期的参数：

accounts/views.py (ch17l040)

```
def login(request):
    auth.authenticate(uid=request.GET.get('token'))
    return redirect('/')
```

现在测试通过了：

```
$ python manage.py test accounts
[...]
Ran 15 tests in 0.041s

OK
```

19.5.1 使用驭件的返回值

接下来，要检查 authenticate 函数是否返回一个用户，供 auth.login 使用。测试像下面这样编写：

accounts/tests/test_views.py (ch17l041)

```
@patch('accounts.views.auth')   ❶
def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth):
    response = self.client.get('/accounts/login?token=abcd123')
    self.assertEqual(
        mock_auth.login.call_args,   ❷
        call(response.wsgi_request, mock_auth.authenticate.return_value)   ❸
    )
```

❶ 还是模拟 contrib.auth 模块。

❷ 这一次检查 auth.login 函数的调用参数。

❸ 检查调用的参数是不是视图收到的请求对象，以及 authenticate 函数返回的“用户”对象。因为 authenticate 也是驭件，所以可以使用特殊的 return_value 属性。

调用驭件的结果是得到另一个驭件。不过，我们也可以从调用的原驭件中获得返回的驭件副本。为了解释清楚，我不得不多次使用“驭件”这个词。下面在控制台中演示一下，希望能帮助你理解：

```
>>> m = Mock()
>>> thing = m()
>>> thing
<Mock name='mock()' id='140652722034952'>
>>> m.return_value
<Mock name='mock()' id='140652722034952'>
>>> thing == m.return_value
True
```

先不管这些，我们想知道测试的结果如何：

```
$ python manage.py test accounts
[...]
```

```
    call(response.wsgi_request, mock_auth.authenticate.return_value)
AssertionError: None != call(<WSGIRequest: GET '/accounts/login?t[...]
```

显然，测试指出我们根本没有调用 auth.login。下面调用它。这一次还是故意做错。

accounts/views.py (ch17l042)

```
def login(request):
    auth.authenticate(uid=request.GET.get('token'))
    auth.login('ack!')
    return redirect('/')
```

调用的确实是 ack!：

```
TypeError: login() missing 1 required positional argument: 'user'
[...]
AssertionError: call('ack!') != call(<WSGIRequest: GET
'/accounts/login?token=[...]
```

下面修正：

accounts/views.py (ch17l043)

```
def login(request):
    user = auth.authenticate(uid=request.GET.get('token'))
    auth.login(request, user)
    return redirect('/')
```

这一次得到了意料之外的失败：

```
ERROR: test_redirects_to_home_page (accounts.tests.test_views.LoginViewTest)
[...]
AttributeError: 'AnonymousUser' object has no attribute '_meta'
```

这是因为我们在所有情况下都调用 auth.login，从而导致针对重定向的测试（目前**没有**模拟 auth.login）出问题。为了修正这个问题，我们要添加一个 if（外加一个测试）。届时，我们将学习如何在类一级上打补丁。

19.5.2 在类一级上打补丁

我们还要编写一个测试，而且也要使用 @patch('accounts.views.auth') 装饰，这样便开始出现重复了。根据“三则重构”原则，可以把 patch 装饰器移到类一级上。这样，测试类中的每一个测试方法都将模拟 accounts.views.auth。不过，这也意味着之前针对重定向的测试也要注入 mock_auth 变量：

accounts/tests/test_views.py (ch17l044)

```
@patch('accounts.views.auth')  ❶
class LoginViewTest(TestCase):

    def test_redirects_to_home_page(self, mock_auth):  ❷
        [...]

    def test_calls_authenticate_with_uid_from_get_request(self, mock_auth):  ❸
```

```
        [...]

    def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth):   ❸
        [...]


    def test_does_not_login_if_user_is_not_authenticated(self, mock_auth):
        mock_auth.authenticate.return_value = None   ❹
        self.client.get('/accounts/login?token=abcd123')
        self.assertEqual(mock_auth.login.called, False)   ❺
```

❶ 把 patch 装饰器移到类一级……

❷ 所以第一个测试方法多了一个参数……

❸ 而且可以删掉其他测试上的装饰器。

❹ 在新测试中，调用 self.client.get **之前**在 auth.authenticate 驭件上设定 return_value。

❺ 下断言，在 authenticate 返回 None 时，不应该调用 auth.login。

现在假失败不见了，得到了意义明确的预期失败：

```
    self.assertEqual(mock_auth.login.called, False)
AssertionError: True != False
```

像这样调整视图，让测试通过：

accounts/views.py (ch17l045)

```
def login(request):
    user = auth.authenticate(uid=request.GET.get('token'))
    if user:
        auth.login(request, user)
    return redirect('/')
```

可以收工了吗？

19.6　关键时刻：功能测试能通过吗

我觉得应该看一下功能测试结果如何了。

下面修改基模板，为已登录用户和未登录用户显示不同的导航栏（功能测试检查的就是这个）：

lists/templates/base.html (ch17l046)

```
<nav class="navbar navbar-default" role="navigation">
  <div class="container-fluid">
    <a class="navbar-brand" href="/">Superlists</a>
    {% if user.email %}
      <ul class="nav navbar-nav navbar-right">
        <li class="navbar-text">Logged in as {{ user.email }}</li>
        <li><a href="#">Log out</a></li>
      </ul>
    {% else %}
```

```
        <form class="navbar-form navbar-right"
              method="POST"
              action="{% url 'send_login_email' %}">
          <span>Enter email to log in:</span>
          <input class="form-control" name="email" type="text" />
          {% csrf_token %}
        </form>
      {% endif %}
    </div>
  </nav>
```

看看测试能否通过：

```
$ python manage.py test functional_tests.test_login
Internal Server Error: /accounts/login
[...]
  File "/.../superlists/accounts/views.py", line 31, in login
    auth.login(request, user)
[...]
ValueError: The following fields do not exist in this model or are m2m fields:
last_login
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Log out
```

噢，不！出问题了。如果 settings.py 中还保留着前面设定的 `LOGGING`，你应该会看到如上所示的详细调用跟踪。可以看出，问题与 `last_login` 字段有关。

我觉得这是 Django 的缺陷，可验证框架就是预期用户模型有个 `last_login` 字段，而我们的用户模型没有。不过，别害怕！解决方法总是有的。

先写一个单元测试重现这个缺陷。因为这与我们自定义的用户模型有关，所以放在 test_models.py 中比较合适：

accounts/tests/test_models.py (ch17l047)

```
from django.test import TestCase
from django.contrib import auth
from accounts.models import Token
User = auth.get_user_model()


class UserModelTest(TestCase):

    def test_user_is_valid_with_email_only(self):
        [...]
    def test_email_is_primary_key(self):
        [...]

    def test_no_problem_with_auth_login(self):
        user = User.objects.create(email='edith@example.com')
        user.backend = ''
        request = self.client.request().wsgi_request
        auth.login(request, user) # 不该抛出异常
```

创建一个请求对象和一个用户，然后把它们传给 `auth.login` 函数。

这个测试会向我们报告错误：

```
    auth.login(request, user)  # 不该抛出异常
[...]
ValueError: The following fields do not exist in this model or are m2m fields:
last_login
```

这个缺陷的具体原因其实跟本书没有多大关系，如果你想追根究底，可以看一下调用跟踪中给出的那几行 Django 源码，再读一下 Django 文档中对信号的说明。

重点是，我们可以像这样修正：

accounts/models.py (ch17l048)

```
import uuid
from django.contrib import auth
from django.db import models

auth.signals.user_logged_in.disconnect(auth.models.update_last_login)


class User(models.Model):
    [...]
```

现在功能测试的结果如何了？

```
$ python manage.py test functional_tests.test_login
[...]
.
 ---------------------------------------------------------------------
Ran 1 test in 3.282s

OK
```

19.7　理论上正常，那么实际呢

哇呜！你能相信吗？我简直不能相信！下面执行 `runserver` 命令，亲自检查一下：

```
$ python manage.py runserver
[...]
Internal Server Error: /accounts/send_login_email
Traceback (most recent call last):
  File "/.../superlists/accounts/views.py", line 20, in send_login_email

ConnectionRefusedError: [Errno 111] Connection refused
```

自己动手检查时，你可能会像我一样遇到一个错误。可能的解决方法有两个。

- 在 settings.py 中重新配置电子邮件。
- 可能要在 shell 中导出电子邮件的密码。

superlists/settings.py (ch17l049)

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'obeythetestinggoat@gmail.com'
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASSWORD')
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

以及：

```
$ export EMAIL_PASSWORD="sekrit"
$ python manage.py runserver
```

然后便能看到如图 19-1 所示的界面。

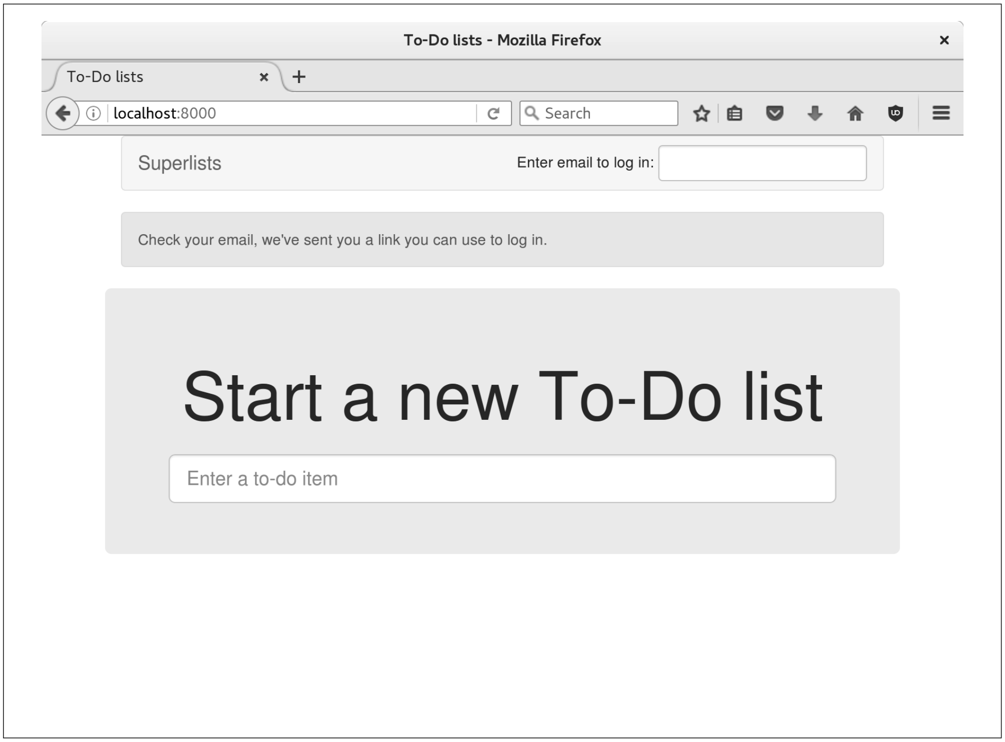


图 19-1：查看你的邮件……

太棒了！

在此之前我一直没提交，因为我在等待一切都能顺利运行的时刻。此时，你可以做一系列单独的提交——登录视图一次、验证后端一次、用户模型一次、修改模板一次。或者，考虑到这些代码都有关联，不能独自运行，也可以做一次大提交：

```
$ git status
$ git add .
$ git diff --staged
$ git commit -m "Custom passwordless auth backend + custom user model"
```

19.8　完善功能测试，测试退出功能

收工之前还有最后一件事要做：测试退出链接。在现有功能测试的基础上添加几步：

functional_tests/test_login.py (ch17l050)

```
        [...]
        # 她登录了！
        self.wait_for(
            lambda: self.browser.find_element_by_link_text('Log out')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn(TEST_EMAIL, navbar.text)

        # 现在她要退出
        self.browser.find_element_by_link_text('Log out').click()

        # 她退出了
        self.wait_for(
            lambda: self.browser.find_element_by_name('email')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn(TEST_EMAIL, navbar.text)
```

这样修改之后，测试会失败，原因是退出按钮没起作用：

```
$ python manage.py test functional_tests.test_login
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [name="email"]
```

实现退出按钮的方法其实很简单：可以使用 Django 内置的退出视图，让它清空用户的会话，然后重定向到我们指定的页面。

accounts/urls.py (ch17l051)

```
from django.contrib.auth.views import logout
[...]

urlpatterns = [
    url(r'^send_login_email$', views.send_login_email, name='send_login_email'),
    url(r'^login$', views.login, name='login'),
    url(r'^logout$', logout, {'next_page': '/'}, name='logout'),
]
```

然后在 base.html 中，让退出按钮指向一个真的 URL：

lists/templates/base.html (ch17l052)

```
    <li><a href="{% url 'logout' %}">Log out</a></li>
```

现在，功能测试都能通过了。其实，整个测试组件都可以通过：

```
$ python manage.py test functional_tests.test_login
[...]
OK
```

```
$ python manage.py test
[...]
Ran 59 tests in 78.124s

OK
```

> 我们离真正安全或可行的登录系统还远着呢！这只是一本书的示例应用，可以就此结束。但是在实际使用中，你得研究很多安全和可用性问题，要做的事还多着呢！这里，我们以身犯险，“自己动手实现加密”——其实依赖现有的登录系统将安全得多。

下一章将充分利用这个登录系统。现在，做次提交，然后阅读总结。

在 Python 中使用模拟技术

- 模拟技术和外部依赖
  编写单元测试时，如果涉及外部依赖，但又不想在测试中真的使用那个依赖，就可以使用模拟技术。驭件用于模拟第三方 API。虽然在 Python 中可以自己创建驭件，但模拟框架（例如 `mock` 模块）可以提供很多便利，让编写测试变得更简单。更重要的是，能让测试读起来更顺口。

- 打猴子补丁
  在运行时替换某个命名空间中的对象。前面的单元测试使用驭件（通过 `patch` 装饰器）替代有额外副作用的真实函数。

- Mock 库
  Michael Foord（在我加入 PythonAnywhere 之前，他在孕育 PythonAnywhere 的公司工作）开发了很优秀的 Mock 库，现在这个库已经集成到 Python 3 的标准库中。这个库包含了在 Python 中使用模拟技术所需的几乎全部功能。

- `patch` 装饰器
  `unittest.mock` 模块提供的 `patch` 函数可用于模拟要测试的模块中的任何一个对象。`patch` 一般用来装饰测试方法，不过也可以放在类一级，应用到类中的所有测试方法上。

- 驭件可能导致与实现紧密耦合
  如前文的旁注所述，驭件可能导致与实现紧密耦合。鉴于此，除非有足够的理由，否则不应该使用驭件。

- 驭件能减少测试中的重复
  而另一方面，在测试中又没必要重复编写使用某个函数的高层级代码。此时使用驭件能减少重复。

接下来还将更为深入地讨论驭件的优缺点。敬请期待！

第20章

测试固件和一个显式等待装饰器

有了一个可以使用的认证系统，现在使用这个系统来识别用户，展示用户创建的所有清单。

为此，要在功能测试中使用已经登录的用户对象。但不能每个测试都走一遍发送登录邮件过程，这么做浪费时间，所以跳过这一步。

这就是分离关注点。功能测试和单元测试的区别在于，前者往往不止有一个断言。但是，理论上一个测试只应该测试一件事，所以没必要在每个功能测试中都测试登录和退出功能。如果能找到一种方法"作弊"，跳过认证，就不用花时间等待执行完重复的测试路径了。

> 在功能测试中去除重复时不要做得过火了。功能测试的优势之一是，可以捕获应用不同部分之间交互时产生的神秘莫测的表现。

> 本章专为这一版重写了。如果遇到问题，或者有改进建议，请通过 obeythetestinggoat@gmail.com 告诉我。

20.1　事先创建好会话，跳过登录过程

用户再次访问网站时 cookie 依然存在，这种现象很常见。也就是说，之前用户已经通过认证了。所以这种"作弊"手段并非异想天开。具体的做法如下：

functional_tests/test_my_lists.py

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
from django.contrib.sessions.backends.db import SessionStore
from .base import FunctionalTest
User = get_user_model()


class MyListsTest(FunctionalTest):

    def create_pre_authenticated_session(self, email):
        user = User.objects.create(email=email)
        session = SessionStore()
        session[SESSION_KEY] = user.pk  ❶
        session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
        session.save()
        ## 为了设定cookie，我们要先访问网站
        ## 而404页面是加载最快的
        self.browser.get(self.live_server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session.session_key,  ❷
            path='/',
        ))
```

❶ 在数据库中创建一个会话对象。会话键的值是用户对象的主键，即用户的电子邮件地址。

❷ 然后把一个 cookie 添加到浏览器中，cookie 的值和服务器中的会话匹配。这样再次访问网站时，服务器就能识别已登录的用户。

注意，这种做法仅在使用 `LiveServerTestCase` 时才有效，所以已创建的 `User` 和 `Session` 对象只存在于测试服务器的数据库中。稍后修改实现的方式，让这个测试也能在过渡服务器里的数据库中运行。

Django 会话：通过 cookie 告知服务器，用户已通过身份验证

我斗胆尝试说明 Django 中的会话、cookie 和身份验证。

HTTP 是无状态的，服务器需要通过某种方式识别**每次请求**是哪个客户端发送的。IP 地址可以共享，因此常用的方案是为每个客户端分配一个唯一的会话 ID，存储在 cookie 中，随每次请求发送。服务器会把这个 ID 存储在某处（默认存储在数据库中），从而识别请求是哪个客户端发送的。

在开发服务器中登录网站后，如果愿意，可以自己动手查看会话 ID，它默认存储在 `sessionid` 键名下，如图 20-1 所示。

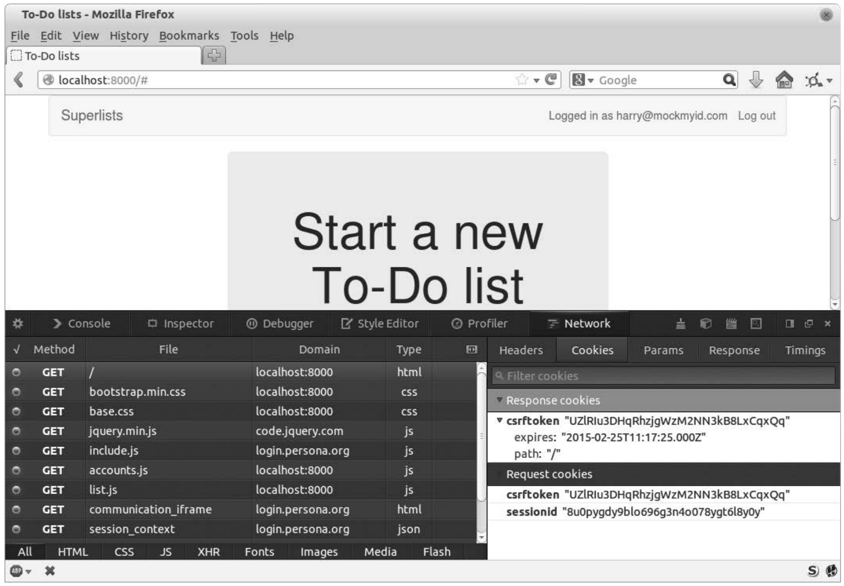


图 20-1：在调试工具中查看会话 cookie

Django 网站会为所有访客设定会话 cookie，不管有没有登录。

为了识别已登录的用户（即通过身份验证），服务器不会让客户端每次请求都发送用户名和密码，而是把客户端的会话标记为已通过验证的会话，并把它与数据库中的用户 ID 关联起来。

会话是类似字典的数据结构，用户 ID 存储在 `django.contrib.auth.SESSION_KEY` 设定的键名下。如果想查看会话的值，可以打开 `./manage.py shell`：

```
$ python manage.py shell
[...]
In [1]: from django.contrib.sessions.models import Session

# 替换成你浏览器cookie中的会话ID
In [2]: session = Session.objects.get(
    session_key="8u0pygdy9blo696g3n4o078ygt6l8y0y"
)

In [3]: print(session.get_decoded())
{'_auth_user_id': 'obeythetestinggoat@gmail.com', '_auth_user_backend':
'accounts.authentication.PasswordlessAuthenticationBackend'}
```

你还可以在用户的会话中存储其他信息，作为一种临时跟踪状态的方式。这对未登录的用户也是有效的。如果想这么做，只需在任意视图中使用 `request.session`，它与字典的操作方式一样。详情参见 Django 文档对会话的说明。

检查是否可行

要检查这种做法是否可行，我们要使用现有测试中的一些代码。下面分别定义两个方法：`wait_to_be_logged_in` 和 `wait_to_be_logged_out`。要想在不同的测试中访问这两个方法，要把它们放到 `FunctionalTest` 类中。此外，我们还得稍微修改一下，让它们可以接收任意的电子邮件地址作为参数：

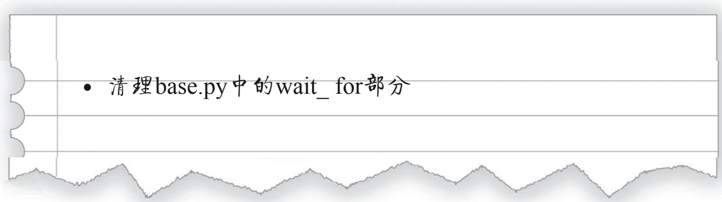
functional_tests/base.py (ch18l002)

```
class FunctionalTest(StaticLiveServerTestCase):
    [...]

    def wait_to_be_logged_in(self, email):
        self.wait_for(
            lambda: self.browser.find_element_by_link_text('Log out')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn(email, navbar.text)


    def wait_to_be_logged_out(self, email):
        self.wait_for(
            lambda: self.browser.find_element_by_name('email')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn(email, navbar.text)
```

嗯，还不错，但我不太喜欢这里重复出现的 `wait_for` 部分。先在便签上记录下来，稍后再修改，让这两个辅助方法可用。



首先，在 test_login.py 中使用它们：

functional_tests/test_login.py (ch18l003)

```
    def test_can_get_email_link_to_log_in(self):
        [...]
        # 她登录了!
        self.wait_to_be_logged_in(email=TEST_EMAIL)

        # 现在她要退出
        self.browser.find_element_by_link_text('Log out').click()

        # 她退出了
        self.wait_to_be_logged_out(email=TEST_EMAIL)
```

为了确认我们没有破坏现有功能，再次运行登录测试：

```
$ python manage.py test functional_tests.test_login
[...]
OK
```

现在可以为“My Lists”页面编写一个占位测试，检查事先创建认证会话的做法是否可行：

functional_tests/test_my_lists.py (ch18l004)

```
    def test_logged_in_users_lists_are_saved_as_my_lists(self):
        email = 'edith@example.com'
        self.browser.get(self.live_server_url)
        self.wait_to_be_logged_out(email)

        # 伊迪丝是已登录用户
        self.create_pre_authenticated_session(email)
        self.browser.get(self.live_server_url)
        self.wait_to_be_logged_in(email)
```

测试的结果为：

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

现在是提交的好时机：

```
$ git add functional_tests
$ git commit -m "test_my_lists: precreate sessions, move login checks into base"
```

JSON 格式的测试固件有危害

使用测试数据预先填充数据库的过程，例如存储 `User` 对象及其相关的 `Session` 对象，叫作设置“测试固件”（test fixture）。

Django 原生支持把数据库中的数据保存为 JSON 格式（使用 `manage.py dumpdata` 命令）。如果在 `TestCase` 中使用类属性 `fixtures`，运行测试时 Django 会自动加载 JSON 格式的数据。

越来越多的人建议不要使用 JSON 格式的固件。如果修改了模型，这种固件维护起来简直就像噩梦一般。此外，对阅读代码的人来说，JSON 固件中众多的属性值让人分不清哪些是对所测试的行为至关重要的，而哪些只是用于补白的。最后，即便从一开始就共用固件，迟早会有测试需要稍微不同的数据，如此一来，为了区分开，只能到处复制整个固件，从而导致无法区分哪些与测试有关，哪些只是恰巧在那儿。

> 通常，直接使用 Django ORM 加载数据要简单得多。如果模型中的字段太多，或者模型之间有关联，即使使用 ORM 也有点烦琐。此时，可以使用一个备受推崇的工具，名为 `factory_boy`。

20.2 显式等待辅助方法最终版：wait装饰器

我们的代码多次用到装饰器。下面我们要自己定义一个，学习装饰器的工作原理。

首先，我们要想好装饰器的作用。我们的计划是，让这个装饰器替代 wait_for_row_in_list_table 中的等待 – 重试 – 超时逻辑，以及 wait_to_be_logged_in/out 中对 self.wait_for 的调用，就像这样：

functional_tests/base.py (ch18l005)

```
    @wait
    def wait_for_row_in_list_table(self, row_text):
        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')
        self.assertIn(row_text, [row.text for row in rows])


    @wait
    def wait_to_be_logged_in(self, email):
        self.browser.find_element_by_link_text('Log out')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn(email, navbar.text)


    @wait
    def wait_to_be_logged_out(self, email):
        self.browser.find_element_by_name('email')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn(email, navbar.text)
```

做好准备了吗？装饰器难以理解（我自己花了好长时间才理解，而且每次自己定义时都要仔细回想），但好消息是我们在 self.wait_for 辅助函数中已经接触过函数式编程了。在函数式编程中，我们把一个函数作为参数传给另一个参数，装饰器也是这个道理。装饰器的不同之处在于，它并不执行代码，而是返回指定函数修改后的版本。

我们这个装饰器要返回一个新函数，它不断调用指定的函数，并捕获常规的异常，直到超时为止。下面是首次尝试：

functional_tests/base.py (ch18l006)

```
def wait(fn):  ❶
    def modified_fn():  ❸
        start_time = time.time()
        while True:  ❹
            try:
                return fn()  ❺
            except (AssertionError, WebDriverException) as e:  ❹
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)
    return modified_fn  ❷
```

❶ 装饰器的作用是修改函数，其参数是一个函数……

❷ ……而返回的则是修改后（或“装饰后”）的版本。

❸ 创建函数的修改版。

❹ 这是我们熟悉的循环，它一直运行着，捕获常规的异常，直到超时为止。

❺ 与之前一样，如果没有异常，立即调用传入的函数，然后返回。

这么定义**基本上**是可以的，但是还不完全正确，不信运行测试看看：

```
$ python manage.py test functional_tests.test_my_lists
[...]
    self.wait_to_be_logged_out(email)
TypeError: modified_fn() takes 0 positional arguments but 2 were given
```

与 `self.wait_for` 不同，这个装饰器依附的函数是有参数的：

functional_tests/base.py

```
    @wait
    def wait_to_be_logged_in(self, email):
        self.browser.find_element_by_link_text('Log out')
```

`wait_to_be_logged_in` 有两个位置参数，分别是 `self` 和 `email`。但是，经装饰替换为 `modified_fn` 之后，参数没有了。我们应该怎么做才能把被装饰的 `fn` 的参数传给 `modified_fn` 呢？

答案涉及一点 Python 魔法，`*args` 和 `**kwargs`，即人们熟知的“变长参数”（我也是刚知道）。

functional_tests/base.py (ch18l007)

```
def wait(fn):
    def modified_fn(*args, **kwargs):   ❶
        start_time = time.time()
        while True:
            try:
                return fn(*args, **kwargs)   ❷
            except (AssertionError, WebDriverException) as e:
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)
    return modified_fn
```

❶ 把 `modified_fn` 的参数设为 `*args` 和 `**kwargs`，以此指定它可以接受任意个位置参数和关键字参数。

❷ 在函数的定义中指定之后，还要把它们传给我们真正要调用的 `fn`。

通过这种方式还可以让装饰器修改函数的参数，但是这里不展开讨论了。现在的重点是，装饰器可用了：

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

你知道真正让人高兴的是什么吗？ `self.wait_for` 辅助函数也可以使用 `wait` 装饰器了，如下所示：

functional_tests/base.py (ch18l008)

```
    @wait
    def wait_for(self, fn):
        return fn()
```

妙啊！现在等待 – 重试逻辑封装到一个地方了，而且还可以灵活设定等待，既可以在功能测试内部调用 `self.wait_for`，也可以在任何辅助函数上使用 `@wait` 装饰器。

下一章将把代码部署到过渡服务器上，并在服务器上使用预先验证身份的会话固件。你将看到，这个过程能帮我们找出一两个 bug。

本章所学

- 装饰器很棒
  通过装饰器，可以抽象不同层次的关注点。本章定义的装饰器可以让测试中的断言不同时等待。

- 谨慎去除功能测试中的重复
  没必要让每个功能测试都测试应用的全部功能。在这一章遇到的情况中，我们想避免在每个需要已验证身份的用户的功能测试中走一遍整个登录流程，所以使用测试固件"作弊"，跳过登录过程。在功能测试中，还可能需要跳过其他过程。不过，我要提醒一下，功能测试的目的是捕获应用不同部分之间交互时的异常表现，所以去除重复时一定要谨慎，别过火了。

- 测试固件
  测试固件指运行测试之前要提前准备好的测试数据，通常是指使用一些数据填充数据库。不过如前所示（创建浏览器的 cookie），也会涉及其他准备工作。

- 避免使用 JSON 固件
  Django 提供的 `dumpdata` 和 `loaddata` 等命令简化了把数据库中的数据导出为 JSON 格式的操作，也可以轻易使用 JSON 格式的数据还原数据库。多数人都不建议使用这种测试固件，因为数据库模式发生变化后这种固件很难维护。请使用 Django ORM，或者 `factory_boy` 这类工具。

第21章

服务器端调试技术

做了这么多事之后，我们要停下来想一想：我们定义了几个用于等待的辅助函数，它们有什么用呢？哦，是等待用户登录的。那用户是如何登录的呢？啊，是通过预先构建已验证身份的用户实现的。

21.1　实践是检验真理的唯一标准：在过渡服务器中捕获最后的问题

这个功能测试在本地运行一切顺利，但在过渡服务器中情况如何呢？部署网站试一下。在这个过程中会遇到一个意料之外的问题（体现了过渡服务器的作用）。为了解决这个问题，要找到在测试服务器中管理数据库的方法：

```
$ cd deploy_tools
$ fab deploy --host=elspeth@superlists-staging.ottg.eu
[...]
```

然后重启 Gunicorn：

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl restart gunicorn-superlists-staging.ottg.eu
```

运行功能测试得到的结果如下：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests

======================================================================
ERROR: test_can_get_email_link_to_log_in
(functional_tests.test_login.LoginTest)
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../functional_tests/test_login.py", line 22, in
  test_can_get_email_link_to_log_in
    self.assertIn('Check your email', body.text)
AssertionError: 'Check your email' not found in 'Server Error (500)'


======================================================================
ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/harry/.../book-example/functional_tests/test_my_lists.py",
  line 34, in test_logged_in_users_lists_are_saved_as_my_lists
    self.wait_to_be_logged_in(email)
  File "/worskpace/functional_tests/base.py", line 42, in wait_to_be_logged_in
    self.browser.find_element_by_link_text('Log out')
    [...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: {"method":"link text","selector":"Log out"}
Stacktrace:
[...]

----------------------------------------------------------------------
Ran 8 tests in 27.933s

FAILED (errors=2)
```

无法登录，不管使用真实的电子邮件系统还是使用已经通过验证的会话都不行。看样子我们全新打造的身份验证系统把服务器搞崩溃了。

下面使用服务器端调试技术找出问题所在。

设置日志

为了记录这个问题，配置 Gunicorn，让它记录日志。在服务器中使用 `vi` 或 `nano` 按照下面的方式调整 Gunicorn 的配置：

server: /etc/systemd/system/gunicorn-superlists-staging.ottg.eu.service

```
ExecStart=/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/bin/gunicorn \
    --bind unix:/tmp/superlists-staging.ottg.eu.socket \
    --capture-output \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application
```

这样配置之后，Gunicorn 会在 ~/sites/$SITENAME 文件夹中保存访问日志和错误日志。

还要确保 settings.py 中仍有 `LOGGING` 设置，这样调试信息才能输送到终端。

superlists/settings.py

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
        },
    },
    'root': {'level': 'INFO'},
}
```

再次重启 Gunicorn，然后运行功能测试，或者手动登录试试。在这些操作执行的过程中，可以使用下面的命令监视日志：

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl restart gunicorn-superlists-staging.ottg.eu
elspeth@server:$ tail -f error.log  # 假设位于 ~/sites/$SITENAME文件夹中
```

你应该会看到类似下面的错误：

```
Internal Server Error: /accounts/send_login_email
Traceback (most recent call last):
  File "/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.6/[...]
    response = wrapped_callback(request, *callback_args, **callback_kwargs)
  File
"/home/elspeth/sites/superlists-staging.ottg.eu/source/accounts/views.py", line
20, in send_login_email
    [email]
[...]
    self.connection.sendmail(from_email, recipients, message.as_bytes(linesep=\r\n))
  File "/usr/lib/python3.6/smtplib.py", line 862, in sendmail
    raise SMTPSenderRefused(code, resp, from_addr)
smtplib.SMTPSenderRefused: (530, b'5.5.1 Authentication Required. Learn more
at\n5.5.1 https://support.google.com/mail/?p=WantAuthError [...]
- gsmtp', noreply@superlists)
```

嗯，Gmail 拒绝发送电子邮件，是吗？可那是为什么呢？哦，原来是因为我们没有告诉服务器密码是什么。

21.2　在服务器上通过环境变量设定机密信息

第 11 章讲过在服务器上设定机密信息的一种方式，那时我们在服务器的文件系统中创建了一个一次性的 Python 文件，然后导入，设定 Django 的 SECRET_KEY 设置。

这几章则在 shell 中使用环境变量存储电子邮件的密码。我们也将在过渡服务器上使用这种方式。可以在 Systemd 的配置文件中设定环境变量：

server: /etc/systemd/system/gunicorn-superlists-staging.ottg.eu.service

```
[Service]
User=elspeth
Environment=EMAIL_PASSWORD=yoursekritpasswordhere
WorkingDirectory=/home/elspeth/sites/superlists-staging.ottg.eu/source
[...]
```

> 在安全方面，使用配置文件有个好处：可以限制权限，只让 root 用户可读。而 Python 源文件做不到这一点。

保存这个文件，然后执行 `daemon-reload` 和 `restart gunicorn` 命令。再次运行功能测试，结果如下：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests

[...]
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_login.py", line 25, in
  test_can_get_email_link_to_log_in
    email = mail.outbox[0]
IndexError: list index out of range

[...]

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: {"method":"link text","selector":"Log out"}
```

`my_lists` 失败没变，但是登录测试提供了更多信息：功能测试向前进了，网站看起来能发送电子邮件了（服务器日志没有显示错误），但是在 `mail.outbox` 中找不到邮件……

21.3 调整功能测试，以便通过POP3测试真实的电子邮件

啊，确实应该如此。功能测试现在在真实的服务器中运行，而不是使用 `LiveServerTestCase`，因此我们不能再检查本地的 `django.mail.outbox` 中有没有发出的邮件了。

首先，在功能测试中要想办法判断是不是运行在过渡服务器中。在 base.py 中，把 `staging_server` 变量绑定到 `self` 上：

functional_tests/base.py (ch18l009)

```
    def setUp(self):
        self.browser = webdriver.Firefox()
        self.staging_server = os.environ.get('STAGING_SERVER')
        if self.staging_server:
           self.live_server_url = 'http://' + self.staging_server
```

然后，构建一个辅助函数，使用 Python 标准库中极其难用的 POP3 客户端从真实的 POP3 电子邮件服务器中获取真实的电子邮件：

functional_tests/test_login.py (ch18l010)

```
import os
import poplib
import re
import time
[...]

    def wait_for_email(self, test_email, subject):
        if not self.staging_server:
            email = mail.outbox[0]
            self.assertIn(test_email, email.to)
            self.assertEqual(email.subject, subject)
            return email.body

        email_id = None
        start = time.time()
        inbox = poplib.POP3_SSL('pop.mail.yahoo.com')
        try:
            inbox.user(test_email)
            inbox.pass_(os.environ['YAHOO_PASSWORD'])
            while time.time() - start < 60:
                # 获取最新的10封邮件
                count, _ = inbox.stat()
                for i in reversed(range(max(1, count - 10), count + 1)):
                    print('getting msg', i)
                    _, lines, __ = inbox.retr(i)
                    lines = [l.decode('utf8') for l in lines]
                    print(lines)
                    if f'Subject: {subject}' in lines:
                        email_id = i
                        body = '\n'.join(lines)
                        return body
                time.sleep(5)
        finally:
            if email_id:
                inbox.dele(email_id)
            inbox.quit()
```

测试时我使用的是 Yahoo 账户，你可以使用任何电子邮件服务，只要支持通过 POP3 访问即可。你要在运行功能测试的控制台中设定 `YAHOO_PASSWORD` 环境变量。

接下来调整功能测试中其他需要改动的地方。首先，针对本地环境和过渡服务器为 `test_email` 变量设定不同的值：

functional_tests/test_login.py (ch18l011-1)

```
@@ -7,7 +7,7 @@ from selenium.webdriver.common.keys import Keys

 from .base import FunctionalTest

-TEST_EMAIL = 'edith@example.com'
+
 SUBJECT = 'Your login link for Superlists'


@@ -33,7 +33,6 @@ class LoginTest(FunctionalTest):
                     print('getting msg', i)
                     _, lines, __ = inbox.retr(i)
                     lines = [l.decode('utf8') for l in lines]
-                    print(lines)
                     if f'Subject: {subject}' in lines:
                         email_id = i
                         body = '\n'.join(lines)
@@ -49,6 +48,11 @@ class LoginTest(FunctionalTest):
         # 伊迪丝访问这个很棒的超级列表网站
         # 第一次注意到导航栏中有“登录”区域
         # 看到要求输入电子邮件地址，她便输入了
+        if self.staging_server:
+            test_email = 'edith.testuser@yahoo.com'
+        else:
+            test_email = 'edith@example.com'
+
         self.browser.get(self.live_server_url)
```

然后使用那个变量，并调用新定义的辅助函数：

functional_tests/test_login.py (ch18l011-2)

```
@@ -54,7 +54,7 @@ class LoginTest(FunctionalTest):
             test_email = 'edith@example.com'

         self.browser.get(self.live_server_url)
-        self.browser.find_element_by_name('email').send_keys(TEST_EMAIL)
+        self.browser.find_element_by_name('email').send_keys(test_email)
         self.browser.find_element_by_name('email').send_keys(Keys.ENTER)

         # 出现一个消息，告诉她邮件已经发出
@@ -64,15 +64,13 @@ class LoginTest(FunctionalTest):
         ))

         # 她查看邮件，看到一个消息
-        email = mail.outbox[0]
-        self.assertIn(TEST_EMAIL, email.to)
-        self.assertEqual(email.subject, SUBJECT)
+        body = self.wait_for_email(test_email, SUBJECT)

         # 邮件中有个URL链接
-        self.assertIn('Use this link to log in', email.body)
-        url_search = re.search(r'http://.+/.+$', email.body)
```

```
+        self.assertIn('Use this link to log in', body)
+        url_search = re.search(r'http://.+/.+$', body)
         if not url_search:
-            self.fail(f'Could not find url in email body:\n{email.body}')
+            self.fail(f'Could not find url in email body:\n{body}')
         url = url_search.group(0)
         self.assertIn(self.live_server_url, url)

@@ -80,11 +78,11 @@ class LoginTest(FunctionalTest):
         self.browser.get(url)

         # 她登录了！
-        self.wait_to_be_logged_in(email=TEST_EMAIL)
+        self.wait_to_be_logged_in(email=test_email)

         # 现在她要退出
         self.browser.find_element_by_link_text('Log out').click()

         # 她退出了
-        self.wait_to_be_logged_out(email=TEST_EMAIL)
+        self.wait_to_be_logged_out(email=test_email)
```

不管你信不信，这样改动之后就行了。经功能测试确认，登录功能可以正常使用了——我们可是发送了真实的电子邮件啊！

我费了好大劲才凑出了检查电子邮件的代码，目前还不雅观，有点脆弱（常见的问题是错误沿用上一次测试的电子邮件）。如果能整理一下，再多试试，我想代码会更可靠。如果不想这么麻烦，可以使用 mailinator.com 这样的服务，只需少量费用就能获得一个一次性的电子邮件地址和查看邮件的 API。

21.4　在过渡服务器中管理测试数据库

现在可以再次运行功能测试，此时又会看到一个失败测试，因为无法创建已经通过认证的会话，所以针对“My Lists”页面的测试失败了：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests

ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
[...]
selenium.common.exceptions.TimeoutException: Message: Could not find element
with id id_logout. Page text was:
Superlists
Sign in
Start a new To-Do list

Ran 8 tests in 72.742s

FAILED (errors=1)
```

失败的真正原因是 `create_pre_authenticated_session` 函数只能操作本地数据库。我们要找到一种方法，管理服务器中的数据库。

21.4.1 创建会话的Django管理命令

若想在服务器中操作，就要编写一个自成一体的脚本，在服务器中的命令行里运行。大多数情况下都会使用 Fabric 执行这样的脚本。

尝试编写可在 Django 环境中运行的独立脚本（和数据库交互等），有些问题要谨慎处理，例如正确设定 `DJANGO_SETTINGS_MODULE` 环境变量，还要正确处理 `sys.path`。

我们可不想在这些细节上浪费时间。其实 Django 允许我们自己创建“管理命令”（可以使用 `python manage.py` 运行的命令），可以把一切琐碎的事情都交给 Django 完成。管理命令保存在应用的 management/commands 文件夹中：

```
$ mkdir -p functional_tests/management/commands
$ touch functional_tests/management/__init__.py
$ touch functional_tests/management/commands/__init__.py
```

管理命令的样板代码是一个类，继承自 `django.core.management.BaseCommand`，而且定义了一个名为 `handle` 的方法：

functional_tests/management/commands/create_session.py

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
User = get_user_model()
from django.contrib.sessions.backends.db import SessionStore
from django.core.management.base import BaseCommand


class Command(BaseCommand):

    def add_arguments(self, parser):
        parser.add_argument('email')

    def handle(self, *args, **options):
        session_key = create_pre_authenticated_session(options['email'])
        self.stdout.write(session_key)


def create_pre_authenticated_session(email):
    user = User.objects.create(email=email)
    session = SessionStore()
    session[SESSION_KEY] = user.pk
    session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
    session.save()
    return session.session_key
```

`create_pre_authenticated_session` 函数的代码从 test_my_lists.py 文件中提取而来。`handle` 方法从选项中获取电子邮件地址，返回一个将要存入浏览器 cookie 中的会话键。这个管理命令还会把会话键打印到命令行中。试一下这个命令：

```
$ python manage.py create_session a@b.com
Unknown command: 'create_session'
```

还要做一步设置——把 `functional_tests` 加入 settings.py，让 Django 把它识别为一个可能包含管理命令和测试的真正应用：

superlists/settings.py

```
+++ b/superlists/settings.py
@@ -42,6 +42,7 @@ INSTALLED_APPS = [
     'lists',
     'accounts',
+    'functional_tests',
 ]
```

现在这个管理命令可以使用了：

```
$ python manage.py create_session a@b.com
qnslckvp2aga7tm6xuivyb0ob1akzzwl
```

如果报错说缺少 `auth_user` 表，可能就要执行 `manage.py migrate` 命令。如果还不行，重新来过，删除 `db.sqlite3`，再执行 `migrate` 命令。

21.4.2　让功能测试在服务器上运行管理命令

接下来调整 test_my_lists.py 文件中的测试，让它在本地服务器中运行本地函数，但是在过渡服务器中运行管理命令：

functional_tests/test_my_lists.py (ch18l016)

```
from django.conf import settings
from .base import FunctionalTest
from .server_tools import create_session_on_server
from .management.commands.create_session import create_pre_authenticated_session

class MyListsTest(FunctionalTest):

    def create_pre_authenticated_session(self, email):
        if self.staging_server:
            session_key = create_session_on_server(self.staging_server, email)
        else:
            session_key = create_pre_authenticated_session(email)
        ## 为了设定cookie，我们要先访问网站
        ## 而404页面是加载最快的
        self.browser.get(self.live_server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session_key,
            path='/',
        ))

    [...]
```

此外，还要调整 base.py，当处在过渡服务器中时，提供更多信息：

functional_tests/base.py (ch18l017)

```
from .server_tools import reset_database  ❶
[...]

class FunctionalTest(StaticLiveServerTestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.staging_server = os.environ.get('STAGING_SERVER')
        if self.staging_server:
            self.live_server_url = 'http://' + self.staging_server
            reset_database(self.staging_server)  ❶
```

❶ 这个函数的作用是在两次测试之间还原服务器中的数据库。稍后我会讲解创建会话这段代码的逻辑，以及为什么可以这么做。

21.4.3 直接在Python代码中使用Fabric

除了 `fab` 命令之外，Fabric 还提供了 API，这样便可以直接在 Python 代码中执行 Fabric 服务器命令。为此，只需通过一个字符串告诉 Fabric 你想连接的主机即可：

functional_tests/server_tools.py

```
from fabric.api import run
from fabric.context_managers import settings


def _get_manage_dot_py(host):
    return f'~/sites/{host}/virtualenv/bin/python ~/sites/{host}/source/manage.py'


def reset_database(host):
    manage_dot_py = _get_manage_dot_py(host)
    with settings(host_string=f'elspeth@{host}'):  ❶
        run(f'{manage_dot_py} flush --noinput')  ❷


def create_session_on_server(host, email):
    manage_dot_py = _get_manage_dot_py(host)
    with settings(host_string=f'elspeth@{host}'):  ❶
        session_key = run(f'{manage_dot_py} create_session {email}')  ❷
        return session_key.strip()
```

❶ 这个上下文管理器设定主机字符串，格式为 `user@server-address`（我直接把自己的服务器用户名 elspeth 写进去了，别忘了修改）。

❷ 在上下文中可以像在 fabfile 中那样直接调用 Fabric 命令。

21.4.4　回顾：在本地服务器和过渡服务器中创建会话的方式

充分理解了吗？下面通过 ASCII 图表回顾一下：

1. 在本地

```
+-----------------------------------+     +-------------------------------------+
| MyListsTest                       | --> | .management.commands.create_session |
| .create_pre_authenticated_session |     |  .create_pre_authenticated_session  |
|           (locally)               |     |           (locally)                 |
+-----------------------------------+     +-------------------------------------+
```

2. 在过渡服务器中

```
+-----------------------------------+     +-------------------------------------+
| MyListsTest                       |     | .management.commands.create_session |
| .create_pre_authenticated_session |     |  .create_pre_authenticated_session  |
|            (locally)              |     |          (on server)                |
+-----------------------------------+     +-------------------------------------+
            |                                                 ^
            v                                                 |
+----------------------------+     +--------+     +------------------------------+
| server_tools               | --> | fabric | --> | ./manage.py create_session   |
| .create_session_on_server  |     | "run"  |     |       (on server)            |
|        (locally)           |     +--------+     +------------------------------+
+----------------------------+
```

不论如何，看一下这么做是否可行。首先，在本地运行测试，确认没有造成任何破坏：

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

然后在服务器中运行。先把代码推送到服务器中：

```
$ git push # 要先提交改动
$ cd deploy_tools
$ fab deploy --host=superlists-staging.ottg.eu
```

再运行测试：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test \
 functional_tests.test_my_lists
[...]
[superlists-staging.ottg.eu] Executing task 'reset_database'
~/sites/superlists-staging.ottg.eu/source/manage.py flush --noinput
[superlists-staging.ottg.eu] out: Syncing...
[superlists-staging.ottg.eu] out: Creating tables ...
[...]
.
 ---------------------------------------------------------------------
Ran 1 test in 25.701s

OK
```

看起来没问题。还可以运行全部测试确认一下：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test \
 functional_tests
[...]
[superlists-staging.ottg.eu] Executing task 'reset_database'
[...]
Ran 8 tests in 89.494s

OK
```

太棒了！

我展示了管理测试数据库的一种方法，你也可以试验其他方式。例如，使用 MySQL 或 Postgres，可以打开一个 SSH 隧道连接服务器，使用端口转发直接操作数据库。然后修改 `settings.DATABASES`，让功能测试使用隧道连接的端口和数据库交互。

警告：小心，不要在线上服务器中运行测试

我们现在遇到一件危险的事，因为编写的代码能直接影响服务器中的数据库。一定要非常非常小心，别在错误的主机中运行功能测试，把生产数据库清空了。

此时，可以考虑使用一些安全防护措施。例如，把过渡环境和生产环境放在不同的服务器中，而且不同的服务器使用不同的口令认证密钥对。

在生产环境的数据副本中运行测试也有同样的危险，我就曾经不小心重复给客户发送了发票（见附录 D）。这是前车之鉴啊。

21.5 集成日志相关的代码

结束本章之前，我们要把日志相关的代码集成到应用中。把输出日志的代码放在那儿，并且纳入版本控制，有助于调试以后遇到的登录问题。毕竟有些人不怀好意。

先把 Gunicorn 的配置保存到 deploy_tools 文件夹里的临时文件中：

deploy_tools/gunicorn-systemd.template.service (ch18l020)

```
[...]
Environment=EMAIL_PASSWORD=SEKRIT
ExecStart=/home/elspeth/sites/SITENAME/virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application
[...]
```

然后在配置笔记中加上几点，提醒自己别忘了在 Gunicorn 配置文件中设定电子邮件密码的环境变量：

deploy_tools/provisioning_notes.md (ch18l021)

```
## Systemd服务

* 参见gunicorn-systemd.template.service
* 把SITENAME替换成具体的域名，例如staging.my-domain.com
* 把SEKRIT替换成电子邮件密码
[...]
```

21.6 小结

在服务器中运行新代码总会让一些缺陷和意料之外的问题显露出来。为了解决这些问题，我们做了很多工作，不过在这个过程中也收获颇多。

我们定义了通用的 `wait` 装饰器，这更符合 Python 习惯，在功能测试中到处都可以使用。我们让测试固件既可以在本地使用，也能在服务器中使用（包括对“真实”电子邮件的测试）。此外，还设定了更牢靠的日志配置。

不过，在部署线上网站之前，最好为用户提供他们真正想要的功能——下一章介绍如何在“My Lists”页面中汇总用户的清单。

在过渡服务器中捕获缺陷时学到的知识

- 固件也要在远程服务器中使用

  在本地运行测试，使用 `LiveServerTestCase` 即可以轻松通过 Django ORM 与测试数据库交互。与过渡服务器中的数据库交互就没这么简单了。解决方法之一是使用 Django 管理命令，如前文所示。不过也可以小心探索，找到适合自己的方法。

- 在服务器中重置数据时要格外小心

  能远程清除服务器中整个数据库的命令极其危险，一定要小心再小心，确保不会意外损坏生产数据。

- 日志对调试服务器中的问题非常重要

  你至少要能看到服务器产生的错误消息。对较为棘手的缺陷来说，还要能得到临时的“调试输出”，保存在某个文件中。

第22章

完成“My Lists”页面：由外而内的TDD

本章我要介绍一种技术，叫“由外而内”的 TDD。一直以来，我们几乎都在使用这种技术。“双循环”TDD 流程就体现了由外而内的思想——先编写功能测试，然后再编写单元测试，其实就是从外部设计系统，再分层编写代码。现在我要明确提出这个概念，再讨论其中涉及的常见问题。

22.1 对立技术：“由内而外”

“由外而内”的对立技术是“由内而外”，接触 TDD 之前，大多数人都凭直觉选择后者。提出一个设计想法之后，有时会自然而然地从最内部、最低层的组件开始实现。

例如，就我们现在面临的问题而言，要想为用户提供一个“My Lists”页面显示已经保存的清单，我们首先会迫不及待地在 `List` 模型中添加 `owner` 属性，因为根据需求推断，显然需要这样一个属性。之后，借助这个新属性修改外层代码，例如视图和模板，最后添加 URL 路由，指向新视图。

这么做感觉更自然，因为所用的代码从来不会依赖尚未实现的功能。内层的一切都是构建外层的坚实基础。

不过，像这样由内而外的工作方式也有缺点。

22.2　为什么选择使用“由外而内”

由内而外的技术最明显的问题是它迫使我们抛开 TDD 流程。功能测试第一次失败可能是因为缺少 URL 路由，但我们决定忽略这一点，先为数据库模型对象添加属性。

我们可能已经在脑海中构思好了内层的模样，而且这些想法往往都很好，不过这些都是对真实需求的推测，因为还未构造使用内层组件的外层组件。

这么做可能会导致内层组件太笼统，或者比真实需求功能更强——不仅浪费了时间，还把项目变得更为复杂。另一种常见的问题是，创建内层组件使用的 API 乍看起来对内部设计而言很合适，但之后会发现并不适用于外层组件。更糟的是，最后你可能会发现内层组件完全无法解决外层组件需要解决的问题。

与此相反，使用由外而内的工作方式，可以在外层组件的基础上构思想从内层组件获取的最佳 API。下面以实例说明如何使用由外而内技术。

22.3　“My Lists”页面的功能测试

编写下面这个功能测试时，我们从能接触到的最外层开始（表现层），然后是视图函数（或叫“控制器”），最后是最内层，在这个例子中是模型代码。

既然 `create_pre_authenticated_session` 函数可以正常使用，那么就可以直接用来编写针对“My Lists”页面的功能测试：

functional_tests/test_my_lists.py (ch19l001-1)

```
    def test_logged_in_users_lists_are_saved_as_my_lists(self):
        # 伊迪丝是已登录用户
        self.create_pre_authenticated_session('edith@example.com')

        # 她访问首页，新建一个清单
        self.browser.get(self.live_server_url)
        self.add_list_item('Reticulate splines')
        self.add_list_item('Immanentize eschaton')
        first_list_url = self.browser.current_url

        # 她第一次看到My Lists链接
        self.browser.find_element_by_link_text('My lists').click()

        # 她看到这个页面中有她创建的清单
        # 而且清单根据第一个待办事项命名
        self.wait_for(
            lambda: self.browser.find_element_by_link_text('Reticulate splines')
        )
        self.browser.find_element_by_link_text('Reticulate splines').click()
        self.wait_for(
            lambda: self.assertEqual(self.browser.current_url, first_list_url)
        )
```

先创建一个包含几个待办事项的清单，然后检查这个列表会出现在新的“My Lists”页面上，而且以清单中的第一个待办事项命名。

接着前面的单元测试，再创建一个清单，确保的确会出现在“My Lists”页面上。与此同时，再检查只有已登录的用户才能看到“My Lists”页面：

functional_tests/test_my_lists.py (ch19l001-2)

```
        [...]
        self.wait_for(
            lambda: self.assertEqual(self.browser.current_url, first_list_url)
        )

        # 她决定再建一个清单试试
        self.browser.get(self.live_server_url)
        self.add_list_item('Click cows')
        second_list_url = self.browser.current_url

        # 在“My Lists”页面，这个新建的清单也显示出来了
        self.browser.find_element_by_link_text('My lists').click()
        self.wait_for(
            lambda: self.browser.find_element_by_link_text('Click cows')
        )
        self.browser.find_element_by_link_text('Click cows').click()
        self.wait_for(
            lambda: self.assertEqual(self.browser.current_url, second_list_url)
        )

        # 她退出后，“My Lists”链接不见了
        self.browser.find_element_by_link_text('Log out').click()
        self.wait_for(lambda: self.assertEqual(
            self.browser.find_elements_by_link_text('My lists'),
            []
        ))
```

上述功能测试使用了一个新的辅助方法，即 `add_list_item`，它抽象了在正确的输入框中输入文本的操作。这个辅助方法在 base.py 中定义：

functional_tests/base.py (ch19l001-3)

```
from selenium.webdriver.common.keys import Keys
[...]

    def add_list_item(self, item_text):
        num_rows = len(self.browser.find_elements_by_css_selector('#id_list_table tr'))
        self.get_item_input_box().send_keys(item_text)
        self.get_item_input_box().send_keys(Keys.ENTER)
        item_number = num_rows + 1
        self.wait_for_row_in_list_table(f'{item_number}: {item_text}')
```

定义好这个辅助方法之后，可以在其他功能测试中像下面这样使用：

functional_tests/test_list_item_validation.py

```
    self.add_list_item('Buy wellies')
```

我觉得使用这个辅助方法之后，功能测试的可读性提高了不少。我改了 6 处，看看你是否与我一样。

运行全部功能测试，做个提交，然后回到我们正在处理的这个功能测试。你看到的第一个错误应该是这样的：

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: My lists
```

22.4　外层：表现层和模板

目前，这个测试失败，报错无法找到“My Lists”链接。这个问题可以在表现层，即 base.html 模板里的导航条中解决。最少量的代码改动如下所示：

lists/templates/base.html (ch19l002-1)

```
{% if user.email %}
  <ul class="nav navbar-nav navbar-left">
    <li><a href="#">My lists</a></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li class="navbar-text">Logged in as {{ user.email }}</li>
    <li><a href="{% url 'logout' %}">Log out</a></li>
  </ul>
```

显然，这个链接没指向任何页面，不过却能解决这个问题，得到下一个失败消息：

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
    lambda: self.browser.find_element_by_link_text('Reticulate splines')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Reticulate splines
```

失败消息指出要构建一个页面，用标题列出一个用户的所有清单。先从简单的开始——一个 URL 和一个占位模板。

可以再次使用由外而内技术，先从表现层开始，只写上地址，其他什么都不做：

lists/templates/base.html (ch19l002-2)

```
<ul class="nav navbar-nav navbar-left">
  <li><a href="{% url 'my_lists' user.email %}">My lists</a></li>
</ul>
```

22.5　下移一层到视图函数（控制器）

这样改还是会得到模板错误，所以要从表现层和 URL 层下移，进入控制器层，即 Django 中的视图函数。

一如既往，先写测试：

lists/tests/test_views.py (ch19l003)

```
class MyListsTest(TestCase):

    def test_my_lists_url_renders_my_lists_template(self):
        response = self.client.get('/lists/users/a@b.com/')
        self.assertTemplateUsed(response, 'my_lists.html')
```

得到的测试结果为：

```
AssertionError: No templates used to render the response
```

然后修正这个问题，不过还在表现层，更准确地说是 urls.py：

lists/urls.py

```
urlpatterns = [
    url(r'^new$', views.new_list, name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
    url(r'^users/(.+)/$', views.my_lists, name='my_lists'),
]
```

修改之后会得到一个测试失败消息，告诉我们移到下一层之后要做什么：

```
AttributeError: module 'lists.views' has no attribute 'my_lists'
```

从表现层移到视图层，再定义一个最简单的占位视图：

lists/views.py (ch19l005)

```
def my_lists(request, email):
    return render(request, 'my_lists.html')
```

以及一个最简单的模板：

lists/templates/my_lists.html

```
{% extends 'base.html' %}

{% block header_text %}My Lists{% endblock %}
```

现在单元测试通过了，但功能测试毫无进展，报错说“My Lists”页面没有显示清单。功能测试希望这些清单可以点击，而且以第一个待办事项命名：

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Reticulate splines
```

22.6　使用由外而内技术，再让一个测试通过

仍然使用功能测试驱动每一步的开发工作。

再次从外层开始，编写模板代码，让“My Lists”页面实现设想的功能。现在，要指定希望从低层获取的 API。

22.6.1 快速重组模板的继承层级

基模板目前没有地方放置新内容了，而且“My Lists”页面不需要新建待办事项表单，所以把表单放到一个块中，需要时才显示：

lists/templates/base.html (ch19l007-1)

```
      <div class="row">
        <div class="col-md-6 col-md-offset-3 jumbotron">
          <div class="text-center">
            <h1>{% block header_text %}{% endblock %}</h1>
            {% block list_form %}
              <form method="POST" action="{% block form_action %}{% endblock %}">
                {{ form.text }}
                {% csrf_token %}
                {% if form.errors %}
                  <div class="form-group has-error">
                    <div class="help-block">{{ form.text.errors }}</div>
                  </div>
                {% endif %}
              </form>
            {% endblock %}
          </div>
        </div>
      </div>
```

lists/templates/base.html (ch19l007-2)

```
        <div class="row">
          <div class="col-md-6 col-md-offset-3">
            {% block table %}
            {% endblock %}
          </div>
        </div>

        <div class="row">
          <div class="col-md-6 col-md-offset-3">
            {% block extra_content %}
            {% endblock %}
          </div>
        </div>

      </div>
      <script src="/static/jquery-3.1.1.min.js"></script>
      [...]
```

22.6.2 使用模板设计API

同时，在 my_lists.html 中覆盖 `list_form` 块，把块中的内容清空：

lists/templates/my_lists.html

```
{% extends 'base.html' %}
```

```
{% block header_text %}My Lists{% endblock %}

{% block list_form %}{% endblock %}
```

然后只在 extra_content 块中编写代码：

lists/templates/my_lists.html

```
[...]

{% block list_form %}{% endblock %}

{% block extra_content %}
    <h2>{{ owner.email }}'s lists</h2>  ❶
    <ul>
        {% for list in owner.list_set.all %}  ❷
            <li><a href="{{ list.get_absolute_url }}">{{ list.name }}</a></li>  ❸
        {% endfor %}
    </ul>
{% endblock %}
```

在这个模板中我们做了几个设计决策，这会对内层代码产生一定影响。

❶ 需要一个名为 owner 的变量，在模板中表示用户。

❷ 想使用 owner.list_set.all 遍历用户创建的清单（我碰巧知道 Django ORM 提供了这个属性）。

❸ 想使用 list.name 获取清单的名字，目前清单以其中的第一个待办事项命名。

> 由外而内的 TDD 有时叫作“一厢情愿式编程”，我想你能看出为什么。我们先编写高层代码，这些代码建立在设想的低层基础之上，可是低层尚未实现。

再次运行功能测试，确认没有造成任何破坏，同时查看有无进展：

```
$ python manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Reticulate splines

 ---------------------------------------------------------------------
Ran 8 tests in 77.613s

FAILED (errors=1)
```

虽然没进展，但至少没造成任何破坏。该提交了：

```
$ git add lists
$ git diff --staged
$ git commit -m "url, placeholder view, and first-cut templates for my_lists"
```

22.6.3 移到下一层：视图向模板中传入什么

现在，视图层要回应需求，为模板层提供所需的对象。这里要提供的是清单属主：

lists/tests/test_views.py (ch19l011)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]
class MyListsTest(TestCase):

    def test_my_lists_url_renders_my_lists_template(self):
        [...]

    def test_passes_correct_owner_to_template(self):
        User.objects.create(email='wrong@owner.com')
        correct_user = User.objects.create(email='a@b.com')
        response = self.client.get('/lists/users/a@b.com/')
        self.assertEqual(response.context['owner'], correct_user)
```

测试结果为：

```
KeyError: 'owner'
```

那就这么修改：

lists/views.py (ch19l012)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def my_lists(request, email):
    owner = User.objects.get(email=email)
    return render(request, 'my_lists.html', {'owner': owner})
```

这样修改之后，新测试通过了，但还是能看到前一个测试导致的错误。只需在这个测试中添加一个用户即可：

lists/tests/test_views.py (ch19l013)

```
    def test_my_lists_url_renders_my_lists_template(self):
        User.objects.create(email='a@b.com')
        [...]
```

现在测试全部通过：

```
OK
```

22.7 视图层的下一个需求：新建清单时应该记录属主

下移到模型层之前，视图层还有一部分代码要用到模型：如果当前用户已经登录网站，需要一种方式把新建的清单指派给一个属主。

初期编写的测试如下所示：

lists/tests/test_views.py (ch19l014)

```
class NewListTest(TestCase):
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)   ❶
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

❶ 为了让测试客户端利用已登录用户的身份发送请求，必须先调用 `force_login()`。

这个测试得到的失败消息如下：

```
AttributeError: 'List' object has no attribute 'owner'
```

为了修正这个问题，可以尝试编写如下代码：

lists/views.py (ch19l015)

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

但这个视图其实解决不了问题，因为还不知道怎么保存清单的属主：

```
    self.assertEqual(list_.owner, user)
AttributeError: 'List' object has no attribute 'owner'
```

抉择时刻：测试失败时是否要移入下一层

为了让这个测试通过，就目前的情况而言，要下移到模型层。但还有一个失败测试，要做的工作太多，现在下移可不明智。

可以采用另一种策略，使用驭件把测试和下层组件更明显地**隔离开**。

一方面，使用驭件要做的工作更多，而且驭件会让测试代码更难读懂。另一方面，如果应用更复杂，外部和内部之间的分层更多，测试就会涉及 3~5 层，在深入最底层实现关键功能之前，这些测试一直处于失败状态。测试无法通过，单就一层而言，我们就无法确定它是否能正常运行，只有等到最底层实现之后才有答案。

你在自己的项目中有可能也会遇到这样的抉择时刻。两种方式都要探讨。先走捷径，放任测试失败不管。下一章再回到这里，探讨如何使用增强隔离的方式。

下面做次提交，并且为这次提交打上**标签**，以便下一章能找到这个位置：

```
$ git commit -am "new_list view tries to assign owner but cant"
$ git tag revisit_this_point_with_isolated_tests
```

22.8　下移到模型层

使用由外而内技术得出了两个需求，需要在模型层实现：其一，想使用 `.owner` 属性为清单指派一个属主；其二，想使用 API `owner.list_set.all` 获取清单的属主。

针对这两个需求，先编写一个测试：

lists/tests/test_models.py (ch19l018)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

class ListModelTest(TestCase):

    def test_get_absolute_url(self):
        [...]

    def test_lists_can_have_owners(self):
        user = User.objects.create(email='a@b.com')
        list_ = List.objects.create(owner=user)
        self.assertIn(list_, user.list_set.all())
```

又得到了一个失败的单元测试：

```
    list_ = List.objects.create(owner=user)
    [...]
TypeError: 'owner' is an invalid keyword argument for this function
```

简单些，把模型写成下面这样：

```
from django.conf import settings
[...]

class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
```

可是我们希望属主可有可无。明确表明需求比含糊其辞强，而且测试还可以作为文档，所以再编写一个测试：

lists/tests/test_models.py (ch19l020)

```
    def test_list_owner_is_optional(self):
        List.objects.create()  # 不该抛出异常
```

正确的模型实现如下所示：

lists/models.py

```
from django.conf import settings
[...]
```

```
class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True, null=True)

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])
```

现在运行测试，会看到以前见过的数据库错误：

```
    return Database.Cursor.execute(self, query, params)
django.db.utils.OperationalError: no such column: lists_list.owner_id
```

因为需要做一次迁移：

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0006_list_owner.py
    - Add field owner to list
```

快成功了，不过还有几个错误：

```
ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f364795ef90>>":
"List.owner" must be a "User" instance.
ERROR: test_can_save_a_POST_request (lists.tests.test_views.NewListTest)

[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f364795ef90>>":
"List.owner" must be a "User" instance.
```

现在回到视图层，做些清理工作。注意，这些错误发生在针对 `new_list` 视图的测试中，而且用户没有登录。仅当用户登录后才应该保存清单的属主。在第 19 章中定义的 `.is_authenticated()` 函数现在有用处了（用户未登录时，Django 使用 `AnonymousUser` 类表示用户，此时 `.is_authenticated()` 函数的返回值始终是 `False`）：

lists/views.py (ch19l023)

```
    if form.is_valid():
        list_ = List()
        if request.user.is_authenticated:
            list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        [...]
```

这样修改之后，测试通过了：

```
$ python manage.py test lists
[...]
.......................................
 ---------------------------------------------------------------------
Ran 39 tests in 0.237s

OK
```

现在是提交的好时机：

```
$ git add lists
$ git commit -m "lists can have owners, which are saved on creation."
```

最后一步：实现模板需要的.name属性

使用的由外而内设计方式还有最后一个需求，即清单根据其中第一个待办事项命名：

lists/tests/test_models.py (ch19l024)

```
    def test_list_name_is_first_item_text(self):
        list_ = List.objects.create()
        Item.objects.create(list=list_, text='first item')
        Item.objects.create(list=list_, text='second item')
        self.assertEqual(list_.name, 'first item')
```

lists/models.py (ch19l025)

```
    @property
    def name(self):
        return self.item_set.first().text
```

你可能无法相信，这样测试就能通过了，而且“My Lists”页面（如图 22-1）也能使用了。

```
$ python manage.py test functional_tests
[...]
Ran 8 tests in 93.819s

OK
```

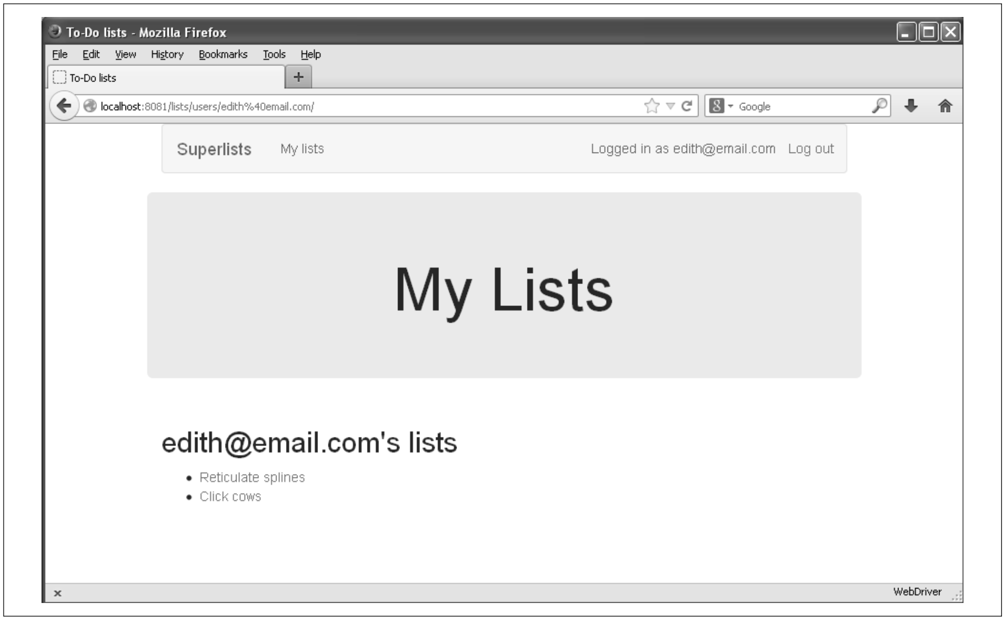


图 22-1：光彩夺目的“My Lists”页面（也证明我在 Windows 中测试过）

Python 中的 `@property` 修饰器

如果你没见过 `@property` 修饰器，我告诉你，它的作用是把类中的方法转变成与属性一样，可以在外部访问。

这是 Python 语言一个强大的特性，因为很容易用它实现“鸭子类型”（duck typing），无须修改类的接口就能改变属性的实现方式。也就是说，如果想把 `.name` 改成模型真正的属性，在数据库中存储文本型数据，整个过程是完全透明的，只要兼顾其他代码，就能继续使用 `.name` 获取清单名，完全不用知道这个属性的具体实现方式。几年前，Raymond Hettinger 在 Pycon 上针对这个话题做过一次出色的演讲。这个演讲对新手友好，我极力推荐你观看（除此之外，还涵盖众多符合 Python 风格的类设计实践方式）。

当然了，就算没使用 `@property` 修饰器，在 Django 的模板语言中还是会调用 `.name` 方法。不过这是 Django 专有的特性，不适用于一般的 Python 程序。

但这个过程中有作弊。测试山羊正满怀猜疑。实现下一层时前一层还有失败的测试。下一章要看一下增强测试隔离性如何编写测试。

由外而内的 TDD

- 由外而内的 TDD
  一种编写代码的方法，由测试驱动，从外层开始（表现层，GUI），然后逐步向内层移动，通过视图层或控制器层，最终达到模型层。这种方法的理念是由实际需要使用的功能驱动代码的编写，而不是在低层猜测需求。

- 一厢情愿式编程
  由外而内的过程有时也叫“一厢情愿式编程”。其实，任何 TDD 形式都涉及一厢情愿。我们总是为还未实现的功能编写测试。

- 由外而内技术的缺点
  由外而内技术也不是万灵药。这种技术鼓励我们关注用户立即就能看到的功能，但不会自动提醒我们为不是那么明显的功能编写关键测试，例如安全相关的功能。你自己要记得编写这些测试。

第 23 章

测试隔离和“倾听测试的心声”

前一章对视图层的失败单元测试放任不管，而是进入模型层编写更多的测试和更多的代码，以便让这个测试通过。

测试侥幸通过了，因为我们的应用很简单。我要强调一点，在复杂的应用中，选择这么做是很危险的。尚未确定高层是否**真正**完成之前就进入低层是一种冒险行为。

> 感谢 Gary Bernhardt，他看了前一章的草稿，建议我深入介绍测试隔离。

确保各层之间相互隔离确实需要投入更多的精力（以及更多可怕的驭件），可是这么做能促使我们得到更好的设计。本章就以实例说明这一点。

23.1　重温抉择时刻：视图层依赖于尚未编写的模型代码

重温前一章的抉择时刻，那时 `new_list` 视图无法正常运行，因为清单还没有 `.owner` 属性。

我们要逆转时光，通过前面做的标签检出以前的代码，看一下使用隔离性更好的测试效果如何。

```
$ git checkout -b more-isolation # 为这次实验新建一个分支
$ git reset --hard revisit_this_point_with_isolated_tests
```

失败的测试是下面这个：

lists/tests/test_views.py

```
class NewListTest(TestCase):
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

尝试使用的解决方法如下所示：

lists/views.py

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

此时，这个视图测试是失败的，因为还未编写模型层：

```
    self.assertEqual(list_.owner, user)
AttributeError: 'List' object has no attribute 'owner'
```

如果没有签出以前的代码并还原 lists/models.py，就不会看到这个错误。这一步一定要做。本章的目标之一是，看一下是否真能为还不存在的模型层编写测试。

23.2　首先尝试使用驭件实现隔离

清单还没有属主，但可以使用一些模拟技术让视图层测试认为有属主：

lists/tests/test_views.py (ch20l003)

```
from unittest.mock import patch
[...]

    @patch('lists.views.List')  ❶
    @patch('lists.views.ItemForm')  ❷
    def test_list_owner_is_saved_if_user_is_authenticated(
        self, mockItemFormClass, mockListClass  ❸
    ):
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
```

```
        self.client.post('/lists/new', data={'text': 'new item'})

        mock_list = mockListClass.return_value  ❹
        self.assertEqual(mock_list.owner, user)  ❺
```

❶ 模拟 `List` 模型的功能，获取视图创建的任何一个清单。

❷ 此外，还要模拟 `ItemForm`。如若不然，调用 `form.save()` 时，表单会抛出错误，因为无法在想创建的 `Item` 对象中使用驭件做外键。一旦开始使用驭件，就很难停手！

❸ 通过测试方法的参数注入驭件时，要按照声明驭件的相反顺序传入。有多个驭件时，方法的签名就是这么奇怪，习惯就好。

❹ 视图访问的清单实例是 `List` 驭件的返回值。

❺ 现在可以声明断言，判断清单对象是否设定了 `.owner` 属性。

现在运行测试应该可以通过：

```
$ python manage.py test lists
[...]
Ran 37 tests in 0.145s
OK
```

如果没通过，确保 views.py 中的视图代码和我前面给出的一模一样，使用的是 `List()`，而不是 `List.objects.create`。

> 使用驭件有个局限，必须按照特定的方式使用 API。这是使用驭件对象要做出的妥协之一。

使用驭件的side_effect属性检查事件发生的顺序

这个测试的问题是，无意中把代码写错也可能侥幸通过测试。假设在指定属主之前不小心调用了 `save` 方法：

lists/views.py

```
    if form.is_valid():
        list_ = List()
        list_.save()
        list_.owner = request.user
        form.save(for_list=list_)
        return redirect(list_)
```

按照测试现在这种写法，它依旧可以通过：

```
OK
```

所以，严格来说，不仅要检查指定了属主，还要确保在清单对象上调用 `save` 方法**之前**就已经指定了。

使用驭件检查事件发生顺序的方法如下。可以模拟一个函数，作为侦件，检查调用这个侦件时周围的状态：

lists/tests/test_views.py (ch20l005)

```
    @patch('lists.views.List')
    @patch('lists.views.ItemForm')
    def test_list_owner_is_saved_if_user_is_authenticated(
        self, mockItemFormClass, mockListClass
    ):

        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        mock_list = mockListClass.return_value

        def check_owner_assigned():  ❶
            self.assertEqual(mock_list.owner, user)
        mock_list.save.side_effect = check_owner_assigned  ❷

        self.client.post('/lists/new', data={'text': 'new item'})

        mock_list.save.assert_called_once_with()  ❸
```

❶ 定义一个函数，在这个函数中就希望先发生的事件声明断言，即检查是否设定了清单的属主。

❷ 把这个检查函数赋值给后续事件的 `side_effect` 属性。当视图在驭件上调用 `save` 方法时，才会执行其中的断言。要保证在测试的目标函数调用前完成此次赋值。

❸ 最后，要确保设定了 `side_effect` 属性的函数一定会被调用，也就是要调用 `.save()` 方法。否则断言永远不会运行。



使用驭件的副作用时有两个常见错误：第一，`side_effect` 属性赋值太晚，也就是在调用测试目标函数**之后**才赋值；第二，忘记检查是否调用了引起副作用的函数。说“常见”，是因为**撰写本章时**我多次同时犯了这两个错误。

现在，如果仍然使用前面有错误的代码，即指定属主和调用 `save` 方法的顺序不对，就会看到如下错误：

```
FAIL: test_list_owner_is_saved_if_user_is_authenticated
(lists.tests.test_views.NewListTest)
[...]
  File "/.../superlists/lists/views.py", line 17, in new_list
    list_.save()
[...]
  File "/.../superlists/lists/tests/test_views.py", line 74, in
check_owner_assigned
    self.assertEqual(mock_list.owner, user)
AssertionError: <MagicMock name='List().owner' id='140691452447208'> != <User:
User object>
```

注意看这个失败消息，它先尝试保存，然后才执行 `side_effect` 属性对应的函数。

可以按照下面的方式修改，让测试通过：

lists/views.py

```
        if form.is_valid():
            list_ = List()
            list_.owner = request.user
            list_.save()
            form.save(for_list=list_)
            return redirect(list_)
```

测试结果为：

```
OK
```

但是这个测试写得很丑！

23.3 倾听测试的心声：丑陋的测试表明需要重构

当你发现必须像这样编写测试，而且要做许多工作时，很有可能表明测试试图向你诉说什么。准备测试所需的数据用了 8 行代码（用户驭件用了 2 行，请求对象又用了 3 行，还有 3 行设定副作用函数），太多了。

这个测试试图告诉我们，视图做的工作太多了，既要创建表单，又要创建清单对象，**还要**决定是否保存清单的属主。

前面已经说过，可以把一部分工作交给表单类完成，把视图变得简单且易于理解一些。为什么要在视图中创建清单对象？或许 `ItemForm.save` 能代劳？为什么视图要决定是否保存 `request.user`？这项任务也可以交给表单完成。

既然要把更多的任务交给表单，感觉应该为这个表单起个新名字。可以叫它 `NewListForm`，因为这个名字能更好地表明它的作用。最终，视图或许可以写成这样吧：

lists/views.py

```
# 先不输入这段代码，只是假设可以这么写

def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user) # 创建List和Item对象
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

这样多简洁！下面来看一下如何为这种写法编写完全隔离的测试。

23.4 以完全隔离的方式重写视图测试

首次尝试为这个视图编写的测试组件**集成度**太高，数据库层和表单层的功能完成之后才能通过。现在使用另一种方式，提高测试的隔离度。

23.4.1　为了新测试的健全性，保留之前的整合测试组件

把 NewListTest 类重名为 NewListViewIntegratedTest，再把尝试使用驭件保存属主的测试代码删掉，换成整合版本，而且暂时为这个测试方法加上 skip 修饰器：

lists/tests/test_views.py (ch20l008)

```
import unittest
[...]

class NewListViewIntegratedTest(TestCase):

    def test_can_save_a_POST_request(self):
        [...]

    @unittest.skip
    def test_list_owner_is_saved_if_user_is_authenticated(self):
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

你听说过“集成测试”（integration test）这个术语吗？想知道它和“整合测试”（integrated test）的区别吗？请看第 26 章框注中的定义。

```
$ python manage.py test lists
[...]
Ran 37 tests in 0.139s
OK
```

23.4.2　完全隔离的新测试组件

从头开始编写测试，看看隔离测试能否驱动我们写出 new_list 视图的替代版本。把这个视图命名为 new_list2，放在旧版视图旁边。写好之后，再换用这个新视图，看看以前的整合测试是否仍然都能通过。

lists/views.py (ch20l009)

```
def new_list(request):
    [...]

def new_list2(request):
    pass
```

23.4.3　站在协作者的角度思考问题

重写测试时若想实现完全隔离，必须丢掉以前对测试的认识。以前我们认为视图的真正作用是操作数据库等，现在则要站在协作对象的角度，思考视图如何与之交互。

站在新的角度上，发现视图的主要协作者是表单对象。所以，为了完全掌控表单，以及按照我们一厢情愿想要的方式定义表单的功能，使用驭件模拟表单。

lists/tests/test_views.py (ch20l010)

```
from unittest.mock import patch
from django.http import HttpRequest
from lists.views import new_list2
[...]

@patch('lists.views.NewListForm')  ❷
class NewListViewUnitTest(unittest.TestCase):  ❶

    def setUp(self):
        self.request = HttpRequest()
        self.request.POST['text'] = 'new list item'  ❸

    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
        new_list2(self.request)
        mockNewListForm.assert_called_once_with(data=self.request.POST)  ❹
```

❶ 使用 Django 提供的 `TestCase` 类太容易写成整合测试。为了确保写出纯粹隔离的单元测试，只能使用 `unittest.TestCase`。

❷ 模拟 `NewListForm` 类（尚未定义）。类中的所有测试方法都会用到这个驭件，所以在类上模拟。

❸ 在 `setUp` 方法中手动创建了一个简单的 POST 请求，没有使用（太过整合的）Django 测试客户端。

❹ 然后检查视图要做的第一件事：在视图中使用正确的构造方法初始化它的协作者，即 `NewListForm`，传入的数据从请求中读取。

在这个测试的结果中首先会看到一个失败消息，报错视图中还没有 `NewListForm`。

```
AttributeError: <module 'lists.views' from '/.../superlists/lists/views.py'>
does not have the attribute 'NewListForm'
```

先编写一个占位表单类：

lists/views.py (ch20l011)

```
from lists.forms import ExistingListItemForm, ItemForm, NewListForm
[...]
```

以及：

lists/forms.py (ch20l012)

```
class ItemForm(forms.models.ModelForm):
    [...]

class NewListForm(object):
    pass

class ExistingListItemForm(ItemForm):
    [...]
```

看到了一个真正的失败消息：

```
AssertionError: Expected 'NewListForm' to be called once. Called 0 times.
```

按照如下的方式编写代码：

lists/views.py (ch20l012-2)

```
def new_list2(request):
    NewListForm(data=request.POST)
```

测试结果为：

```
$ python manage.py test lists
[...]
Ran 38 tests in 0.143s
OK
```

继续编写测试。如果表单中的数据有效，要在表单对象上调用 save 方法：

lists/tests/test_views.py (ch20l013)

```
from unittest.mock import patch, Mock
[...]

@patch('lists.views.NewListForm')
class NewListViewUnitTest(unittest.TestCase):

    def setUp(self):
        self.request = HttpRequest()
        self.request.POST['text'] = 'new list item'
        self.request.user = Mock()


    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
        new_list2(self.request)
        mockNewListForm.assert_called_once_with(data=self.request.POST)


    def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = True
        new_list2(self.request)
        mock_form.save.assert_called_once_with(owner=self.request.user)
```

据此，可以写出如下视图：

lists/views.py (ch20l014)

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    form.save(owner=request.user)
```

如果表单中的数据有效，让视图做一个重定向，把我们带到一个页面，查看表单刚刚创建的对象。所以要模拟视图的另一个协作者——redirect 函数：

lists/tests/test_views.py (ch20l015)

```
    @patch('lists.views.redirect')  ❶
    def test_redirects_to_form_returned_object_if_form_valid(
        self, mock_redirect, mockNewListForm   ❷
    ):
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = True   ❸

        response = new_list2(self.request)

        self.assertEqual(response, mock_redirect.return_value)   ❹
        mock_redirect.assert_called_once_with(mock_form.save.return_value)   ❺
```

❶ 模拟 `redirect` 函数，不过这次直接在方法上模拟。

❷ `patch` 修饰器先应用最内层的那个，所以这个驭件在 `mockNewListForm` 之前传入方法。

❸ 指定测试的是表单中数据有效的情况。

❹ 检查视图的响应是否为 `redirect` 函数的结果。

❺ 然后检查调用 `redirect` 函数时传入的参数是否为在表单上调用 `save` 方法得到的对象。

据此，可以编写如下视图：

lists/views.py (ch20l016)

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    list_ = form.save(owner=request.user)
    return redirect(list_)
```

测试结果为：

```
$ python manage.py test lists
[...]
Ran 40 tests in 0.163s
OK
```

然后测试表单提交失败的情况——如果表单中的数据无效，渲染首页的模板：

lists/tests/test_views.py (ch20l017)

```
    @patch('lists.views.render')
    def test_renders_home_template_with_form_if_form_invalid(
        self, mock_render, mockNewListForm
    ):
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = False

        response = new_list2(self.request)

        self.assertEqual(response, mock_render.return_value)
        mock_render.assert_called_once_with(
            self.request, 'home.html', {'form': mock_form}
        )
```

测试的结果为：

```
AssertionError: <HttpResponseRedirect status_code=302, "te[114 chars]%3E"> !=
<MagicMock name='render()' id='140244627467408'>
```

> 在驭件上调用断言方法时一定要运行测试，确认它会失败。因为输入断言函数时太容易出错，会导致调用的模拟方法没有任何作用。（我会写成 `asssert_called_once_with`，用了三个 s。你自己可以试一下！）

故意犯个错误，确保测试全面：

lists/views.py (ch20l018)

```
    def new_list2(request):
        form = NewListForm(data=request.POST)
        list_ = form.save(owner=request.user)
        if form.is_valid():
            return redirect(list_)
        return render(request, 'home.html', {'form': form})
```

测试本不应该通过却通过了！那就再写一个测试：

lists/tests/test_views.py (ch20l019)

```
    def test_does_not_save_if_form_invalid(self, mockNewListForm):
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = False
        new_list2(self.request)
        self.assertFalse(mock_form.save.called)
```

这个测试会失败：

```
    self.assertFalse(mock_form.save.called)
AssertionError: True is not false
```

最终得到了一个精简的视图：

lists/views.py

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
        return redirect(list_)
    return render(request, 'home.html', {'form': form})
```

测试结果为：

```
$ python manage.py test lists
[...]
Ran 42 tests in 0.163s
OK
```

23.5 下移到表单层

已经写好了视图函数，这个视图基于设想的表单 `NewItemForm`，而且这个表单现在还不存在。

需要在表单对象上调用 `save` 方法创建一个新清单，还要使用通过验证的 POST 数据创建一个新待办事项。如果直接使用 ORM，`save` 方法可以写成这样：

```
class NewListForm(models.Form):

    def save(self, owner):
        list_ = List()
        if owner:
            list_.owner = owner
        list_.save()
        item = Item()
        item.list = list_
        item.text = self.cleaned_data['text']
        item.save()
```

这种实现方式依赖于模型层的两个类，即 `Item` 和 `List`。那么隔离性好的测试应该怎么写呢？

```
class NewListFormTest(unittest.TestCase):

    @patch('lists.forms.List')  ❶
    @patch('lists.forms.Item')  ❶
    def test_save_creates_new_list_and_item_from_post_data(
        self, mockItem, mockList  ❶
    ):
        mock_item = mockItem.return_value
        mock_list = mockList.return_value
        user = Mock()
        form = NewListForm(data={'text': 'new item text'})
        form.is_valid()  ❷

        def check_item_text_and_list():
            self.assertEqual(mock_item.text, 'new item text')
            self.assertEqual(mock_item.list, mock_list)
            self.assertTrue(mock_list.save.called)
        mock_item.save.side_effect = check_item_text_and_list  ❸

        form.save(owner=user)

        self.assertTrue(mock_item.save.called)  ❹
```

❶ 为表单模拟两个来自下部模型层的协作者。

❷ 必须调用 `is_valid()` 方法，这样表单才会把通过验证的数据存储到 `.cleaned_data` 字典中。

❸ 使用 `side_effect` 方法确保保存新待办事项对象时，使用已经保存的清单，而且待办事项中的文本正确。

❹ 一如既往，再次确认调用了副作用函数。

唉，这个测试写得好丑！

始终倾听测试的心声：从应用中删除ORM代码

测试又在诉说什么：Django ORM 很难模拟，而且表单类需要较深入地了解 ORM 的工作方式。再次使用一厢情愿式编程，想想表单想用什么样的简单 API 呢？下面这种怎么样：

```
def save(self):
    List.create_new(first_item_text=self.cleaned_data['text'])
```

又冒出个想法：要不在 `List` 类[1] 中定义一个辅助函数，封装保存新清单对象及相关的第一个待办事项这部分逻辑。

那就先为这个想法编写测试：

lists/tests/test_forms.py (ch20l021)

```
import unittest
from unittest.mock import patch, Mock
from django.test import TestCase

from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm, NewListForm
)
from lists.models import Item, List
[...]


class NewListFormTest(unittest.TestCase):

    @patch('lists.forms.List.create_new')
    def test_save_creates_new_list_from_post_data_if_user_not_authenticated(
        self, mock_List_create_new
    ):
        user = Mock(is_authenticated=False)
        form = NewListForm(data={'text': 'new item text'})
        form.is_valid()
        form.save(owner=user)
        mock_List_create_new.assert_called_once_with(
            first_item_text='new item text'
        )
```

既然已经测试了这种情况，那就再写个测试检查用户已经通过认证的情况吧：

lists/tests/test_forms.py (ch20l022)

```
    @patch('lists.forms.List.create_new')
    def test_save_creates_new_list_with_owner_if_user_authenticated(
        self, mock_List_create_new
    ):
        user = Mock(is_authenticated=True)
        form = NewListForm(data={'text': 'new item text'})
```

注 1：你很可能想定义一个单独的函数，但是放在类中有利于记住它在哪儿。更重要的是，还能表明这个函数的作用。嗯，我保证等我写完这本书之后会这么做的。尖锐的批评就此打住吧！

```
        form.is_valid()
        form.save(owner=user)
        mock_List_create_new.assert_called_once_with(
            first_item_text='new item text', owner=user
        )
```

可以看出，这个测试易读多了。下面开始实现新表单。先从 import 语句开始：

lists/forms.py (ch20l023)

```
from lists.models import Item, List
```

此时驭件说要定义一个占位的 create_new 方法：

```
AttributeError: <class 'lists.models.List'> does not have the attribute
'create_new'
```

lists/models.py

```
class List(models.Model):

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])

    def create_new():
        pass
```

几步之后，最终写出如下的 save 方法：

lists/forms.py (ch20l025)

```
class NewListForm(ItemForm):

    def save(self, owner):
        if owner.is_authenticated:
            List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
        else:
            List.create_new(first_item_text=self.cleaned_data['text'])
```

而且测试也通过了：

```
$ python manage.py test lists
Ran 44 tests in 0.192s
OK
```

把 ORM 代码放到辅助方法中

从编写隔离测试的过程中我们学会了一项技能——“ORM 辅助方法”。

使用 Django 的 ORM 可以通过十分易读的句法（肯定比纯 SQL 好得多）快速完成工作。但有些人喜欢尽量减少应用中使用的 ORM 代码量，尤其不喜欢在视图层和表单层使用 ORM 代码。

一个原因是，测试这几层时更容易；另一个原因是，必须定义辅助方法，这样能更清晰地表述域逻辑。请对比这段代码：

```
            list_ = List()
            list_.save()
            item = Item()
            item.list = list_
            item.text = self.cleaned_data['text']
            item.save()
```

和这段代码：

```
        List.create_new(first_item_text=self.cleaned_data['text'])
```

辅助方法同样可用于读写查询。假设有这样一行代码：

```
        Book.objects.filter(in_print=True, pub_date__lte=datetime.today())
```

和如下的辅助方法相比，孰好孰坏一目了然：

```
        Book.all_available_books()
```

定义辅助方法时，可以起个适当的名字，表明它们在业务逻辑中的作用。使用辅助方法不仅可以让代码的条理变得更清晰，还能把所有 ORM 调用都放在模型层，因此整个应用不同部分之间的耦合更松散。

23.6　下移到模型层

在模型层不用再编写隔离测试了，因为模型层的目的就是与数据库结合在一起工作，所以编写整合测试更合理：

lists/tests/test_models.py (ch20l026)

```
class ListModelTest(TestCase):

    def test_get_absolute_url(self):
        list_ = List.objects.create()
        self.assertEqual(list_.get_absolute_url(), f'/lists/{list_.id}/')


    def test_create_new_creates_list_and_first_item(self):
        List.create_new(first_item_text='new item text')
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'new item text')
        new_list = List.objects.first()
        self.assertEqual(new_item.list, new_list)
```

测试的结果为：

```
TypeError: create_new() got an unexpected keyword argument 'first_item_text'
```

根据测试结果，可以先把实现方式写成这样：

lists/models.py (ch20l027)

```
class List(models.Model):

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])

    @staticmethod
    def create_new(first_item_text):
        list_ = List.objects.create()
        Item.objects.create(text=first_item_text, list=list_)
```

注意，一路走下来，直到模型层，由视图层和表单层驱动，得到了一个设计良好的模型，但是 `List` 模型还不支持属主。

现在，测试清单应该有一个属主。添加如下测试：

lists/tests/test_models.py (ch20l028)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

    def test_create_new_optionally_saves_owner(self):
        user = User.objects.create()
        List.create_new(first_item_text='new item text', owner=user)
        new_list = List.objects.first()
        self.assertEqual(new_list.owner, user)
```

既然已经打开这个文件，那就再为 `owner` 属性编写一些测试吧：

lists/tests/test_models.py (ch20l029)

```
class ListModelTest(TestCase):
    [...]

    def test_lists_can_have_owners(self):
        List(owner=User())  # 不该抛出异常


    def test_list_owner_is_optional(self):
        List().full_clean()  # 不该抛出异常
```

这两个测试和前一章使用的测试几乎一样，不过我稍微改了些，不让它们保存对象。因为对这个测试而言，内存中有这些对象就行了。

> 尽量多用内存中（未保存）的模型对象，这样测试运行得更快。

测试的结果为：

```
$ python manage.py test lists
[...]
ERROR: test_create_new_optionally_saves_owner
TypeError: create_new() got an unexpected keyword argument 'owner'
[...]
ERROR: test_lists_can_have_owners (lists.tests.test_models.ListModelTest)
TypeError: 'owner' is an invalid keyword argument for this function
[...]
Ran 48 tests in 0.204s
FAILED (errors=2)
```

然后按照前一章使用的方式实现模型：

lists/models.py (ch20l030-1)

```
from django.conf import settings
[...]


class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True, null=True)
    [...]
```

此时，测试的结果中有各种完整性失败，执行迁移后才能解决这些问题：

```
django.db.utils.OperationalError: no such column: lists_list.owner_id
```

执行迁移后再运行测试，会看到下面三个失败：

```
ERROR: test_create_new_optionally_saves_owner
TypeError: create_new() got an unexpected keyword argument 'owner'
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f5b2380b4e0>>":
"List.owner" must be a "User" instance.
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f5b237a12e8>>":
"List.owner" must be a "User" instance.
```

先处理第一个失败。这个失败由 `create_new` 方法导致：

lists/models.py (ch20l030-3)

```
    @staticmethod
    def create_new(first_item_text, owner=None):
        list_ = List.objects.create(owner=owner)
        Item.objects.create(text=first_item_text, list=list_)
```

回到视图层

视图层以前的两个整合测试失败了，怎么回事呢？

```
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7fbad1cb6c10>>":
"List.owner" must be a "User" instance.
```

啊，原来是因为以前的视图没有分清谁才是清单的属主：

lists/views.py

```
        if form.is_valid():
            list_ = List()
            list_.owner = request.user
            list_.save()
```

这一刻才意识到以前的代码没有满足需求。修正这个问题，让所有测试都通过：

lists/views.py (ch20l031)

```
    def new_list(request):
        form = ItemForm(data=request.POST)
        if form.is_valid():
            list_ = List()
            if request.user.is_authenticated:
                list_.owner = request.user
            list_.save()
            form.save(for_list=list_)
            return redirect(list_)
        else:
            return render(request, 'home.html', {"form": form})


    def new_list2(request):
        [...]
```

整合测试的好处之一是，可以捕获这种无法轻易预测的交互。我们忘了编写测试检查用户没有通过验证的情况，可是整合测试会由上而下使用整个组件，最终模型层出现了错误，提醒我们忘了一些事。

```
$ python manage.py test lists
[...]
Ran 48 tests in 0.175s
OK
```

23.7 关键时刻，以及使用模拟技术的风险

换掉以前的视图，使用新视图试试。调换视图可以在 urls.py 中完成：

lists/urls.py

```
    [...]
        url(r'^new$', views.new_list2, name='new_list'),
```

还得删除整合测试类上的 `unittest.skip` 修饰器，看看为清单属主编写的新代码是否真得可用：

lists/tests/test_views.py (ch20l033)

```
class NewListViewIntegratedTest(TestCase):

    def test_can_save_a_POST_request(self):
        [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        [...]
        self.assertEqual(list_.owner, user)
```

那么测试的结果如何呢？啊，情况不妙！

```
ERROR: test_list_owner_is_saved_if_user_is_authenticated
[...]
ERROR: test_can_save_a_POST_request
[...]
ERROR: test_redirects_after_POST
(lists.tests.test_views.NewListViewIntegratedTest)
  File "/.../superlists/lists/views.py", line 30, in new_list2
    return redirect(list_)
[...]
TypeError: argument of type 'NoneType' is not iterable

FAILED (errors=3)
```

测试隔离有个很重要的知识点：虽然它有可能帮助你为单独各层做出好的设计，但无法自动验证各层**之间**的集成情况。

上述测试结果表明，视图期望表单返回一个待办事项：

lists/views.py

```
        list_ = form.save(owner=request.user)
        return redirect(list_)
```

但没让表单返回任何值：

lists/forms.py

```
    def save(self, owner):
        if owner.is_authenticated:
            List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
        else:
            List.create_new(first_item_text=self.cleaned_data['text'])
```

23.8 把层与层之间的交互当作“合约”

除了隔离的单元测试之外，就算什么都没写，功能测试最终也能发现这个失误。但理想情况下，我们希望尽早得到反馈——功能测试可能要运行好几分钟，应用变大之后甚至可能要几个小时。在这种问题发生之前有没有办法避免呢？

理论上讲，有办法：把层与层之间的交互看成一种“合约”。只要模拟一层的行为，就要在

心里记住，层与层之间现在有了隐形合约，这一层的驭件或许可以转移到下一层的测试中。遗忘的合约如下所示：

lists/tests/test_views.py

```
    @patch('lists.views.redirect')
    def test_redirects_to_form_returned_object_if_form_valid(
        self, mock_redirect, mockNewListForm
    ):
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = True

        response = new_list2(self.request)

        self.assertEqual(response, mock_redirect.return_value)
        mock_redirect.assert_called_once_with(mock_form.save.return_value)  ❶
```

❶ 模拟的 `form.save` 方法返回一个对象，我们希望在视图中使用这个对象。

23.8.1　找出隐形合约

现在要审查 `NewListViewUnitTest` 类中的每个测试，看看各驭件在隐形合约中表述了什么：

lists/tests/test_views.py

```
    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
        [...]
        mockNewListForm.assert_called_once_with(data=self.request.POST)  ❶


    def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = True  ❷
        new_list2(self.request)
        mock_form.save.assert_called_once_with(owner=self.request.user)  ❸


    def test_does_not_save_if_form_invalid(self, mockNewListForm):
        [...]
        mock_form.is_valid.return_value = False  ❷
        [...]


    @patch('lists.views.redirect')
    def test_redirects_to_form_returned_object_if_form_valid(
        self, mock_redirect, mockNewListForm
    ):
        [...]
        mock_redirect.assert_called_once_with(mock_form.save.return_value)  ❹


    @patch('lists.views.render')
    def test_renders_home_template_with_form_if_form_invalid(
        [...]
```

❶ 需要传入 POST 请求中的数据，以便初始化表单。

❷ 表单对象要能响应 `is_valid()` 方法，而且要根据输入值判断返回 `True` 还是 `False`。

❸ 表单对象要能响应 `.save` 方法，而且传入的参数值是 `request.user`，然后根据用户是否登录做相应处理。

❹ 表单对象的 `.save` 方法应该返回一个新清单对象，以便视图把用户重定向到显示这个对象的页面。

仔细分析表单测试，可以看出，其实只明确测试了❸。❶和❷很幸运，因为这是 Django 中 `ModelForm` 的默认特性，而且针对父类 `ItemForm` 的测试涵盖了这两点。

但❹却成了漏网之鱼。

> 使用由外而内的 TDD 技术编写隔离测试时，要记住每个测试在合约中对下一层应该实现的功能做出的隐含假设，而且记得稍后要回来测试这些假设。你可以在便签上记下来，也可以使用 `self.fail` 编写占位测试。

23.8.2 修正由于疏忽导致的问题

下面添加一个新测试，确保表单返回刚刚保存的清单：

lists/tests/test_forms.py (ch20l038-1)

```
    @patch('lists.forms.List.create_new')
    def test_save_returns_new_list_object(self, mock_List_create_new):
        user = Mock(is_authenticated=True)
        form = NewListForm(data={'text': 'new item text'})
        form.is_valid()
        response = form.save(owner=user)
        self.assertEqual(response, mock_List_create_new.return_value)
```

其实，这是个很好的示例——和 `List.create_new` 之间有隐形合约，希望这个方法返回刚创建的清单对象。下面为这个需求添加一个占位测试：

lists/tests/test_models.py (ch20l038-2)

```
class ListModelTest(TestCase):
    [...]

    def test_create_returns_new_list_object(self):
        self.fail()
```

得到一个失败测试，告诉我们要修正表单对象的 `save` 方法：

```
AssertionError: None != <MagicMock name='create_new()' id='139802647565536'>
FAILED (failures=2, errors=3)
```

修正方法如下：

lists/forms.py (ch20l039-1)

```
class NewListForm(ItemForm):

    def save(self, owner):
        if owner.is_authenticated:
            return List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
        else:
            return List.create_new(first_item_text=self.cleaned_data['text'])
```

这才刚开始。下面应该看一下占位测试：

```
[...]
FAIL: test_create_returns_new_list_object
    self.fail()
AssertionError: None

FAILED (failures=1, errors=3)
```

编写这个测试：

lists/tests/test_models.py (ch20l039-2)

```
    def test_create_returns_new_list_object(self):
        returned = List.create_new(first_item_text='new item text')
        new_list = List.objects.first()
        self.assertEqual(returned, new_list)
```

测试结果为：

```
AssertionError: None != <List: List object>
```

然后加上返回值：

lists/models.py (ch20l039-3)

```
    @staticmethod
    def create_new(first_item_text, owner=None):
        list_ = List.objects.create(owner=owner)
        Item.objects.create(text=first_item_text, list=list_)
        return list_
```

现在整个测试组件都可以通过了：

```
$ python manage.py test lists
[...]
Ran 50 tests in 0.169s

OK
```

23.9　还缺一个测试

以上就是由测试驱动开发出来的保存清单属主功能，这个功能可以正常使用。不过，功能测试却无法通过：

```
$ python manage.py test functional_tests.test_my_lists
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Reticulate splines
```

失败的原因是有一个功能没实现，即清单对象的 `.name` 属性。这里还可以使用前一章的测试和代码：

lists/tests/test_models.py (ch20l040)

```
    def test_list_name_is_first_item_text(self):
        list_ = List.objects.create()
        Item.objects.create(list=list_, text='first item')
        Item.objects.create(list=list_, text='second item')
        self.assertEqual(list_.name, 'first item')
```

（再次说明，因为这是模型层测试，所以使用 ORM 没问题。你可能想使用驭件编写这个测试，不过这么做没什么意义。）

lists/models.py (ch20l041)

```
    @property
    def name(self):
        return self.item_set.first().text
```

现在功能测试可以通过了：

```
$ python manage.py test functional_tests.test_my_lists

Ran 1 test in 21.428s

OK
```

23.10　清理：保留哪些整合测试

现在一切都可以正常运行了，要删除一些多余的测试，还要决定是否保留以前的整合测试。

23.10.1　删除表单层多余的代码

可以把以前针对 `ItemForm` 类中 `save` 方法的测试删掉：

lists/tests/test_forms.py

```
--- a/lists/tests/test_forms.py
+++ b/lists/tests/test_forms.py
@@ -23,14 +23,6 @@ class ItemFormTest(TestCase):

         self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])


-    def test_form_save_handles_saving_to_a_list(self):
-        list_ = List.objects.create()
-        form = ItemForm(data={'text': 'do me'})
-        new_item = form.save(for_list=list_)
```

```
-        self.assertEqual(new_item, Item.objects.first())
-        self.assertEqual(new_item.text, 'do me')
-        self.assertEqual(new_item.list, list_)
-
```

对应用的代码而言，可以把 forms.py 中两个多余的 save 方法删掉：

lists/forms.py

```
--- a/lists/forms.py
+++ b/lists/forms.py
@@ -22,11 +22,6 @@ class ItemForm(forms.models.ModelForm):

         self.fields['text'].error_messages['required'] = EMPTY_ITEM_ERROR

-    def save(self, for_list):
-        self.instance.list = for_list
-        return super().save()
-
-

 class NewListForm(ItemForm):

@@ -52,8 +47,3 @@ class ExistingListItemForm(ItemForm):

             e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
             self._update_errors(e)
-
-
-    def save(self):
-        return forms.models.ModelForm.save(self)
-
```

23.10.2　删除以前实现的视图

现在，可以把以前的 new_list 视图完全删掉，再把 new_list2 重命名为 new_list：

lists/tests/test_views.py

```
-from lists.views import new_list, new_list2
+from lists.views import new_list


 class HomePageTest(TestCase):
@@ -75,7 +75,7 @@ class NewListViewIntegratedTest(TestCase):
         request = HttpRequest()
         request.user = User.objects.create(email='a@b.com')
         request.POST['text'] = 'new list item'
-        new_list2(request)
+        new_list(request)
         list_ = List.objects.first()
         self.assertEqual(list_.owner, request.user)

@@ -91,21 +91,21 @@ class NewListViewUnitTest(unittest.TestCase):
```

```
    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
-        new_list2(self.request)
+        new_list(self.request)

[.. several more]
```

lists/urls.py

```
--- a/lists/urls.py
+++ b/lists/urls.py
@@ -3,7 +3,7 @@ from django.conf.urls import url
 from lists import views

 urlpatterns = [
-    url(r'^new$', views.new_list2, name='new_list'),
+    url(r'^new$', views.new_list, name='new_list'),
     url(r'^(\d+)/$', views.view_list, name='view_list'),
     url(r'^users/(.+)/$', views.my_lists, name='my_lists'),
 ]
```

lists/views.py (ch20l047)

```
def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
        [...]
```

然后检查所有测试是否仍能通过：

```
OK
```

23.10.3　删除视图层多余的代码

最后要决定保留哪些整合测试（如果需要保留的话）。

一种方法是全部删除，让功能测试捕获集成问题。这么做完全可行。

不过，从前文得知，如果在集成各层时犯了小错误，整合测试可以提醒你。可以保留部分测试，作为完整性检查，以便得到快速反馈。

要不就保留下面这三个测试吧：

lists/tests/test_views.py (ch20l048)

```
class NewListViewIntegratedTest(TestCase):

    def test_can_save_a_POST_request(self):
        self.client.post('/lists/new', data={'text': 'A new list item'})
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')
```

```
    def test_for_invalid_input_doesnt_save_but_shows_errors(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(List.objects.count(), 0)
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))


    def test_list_owner_is_saved_if_user_is_authenticated(self):
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

如果最终决定保留中间层的测试，我认为这三个不错，因为我觉得它们涵盖了大部分集成操作：测试了整个组件，从请求直到数据库，而且覆盖了视图最重要的三个用例。

23.11　总结：什么时候编写隔离测试，什么时候编写整合测试

Django 提供的测试工具为快速编写整合测试提供了便利。测试运行程序能帮助我们创建一个存在于内存中的数据库，运行速度很快，而且在两次测试之间还能重建数据库。使用 `TestCase` 类和测试客户端测试视图很简单，可以检查是否修改了数据库中的对象，确认 URL 映射是否可用，还能检查渲染模板的情况。这些工具降低了测试的门槛，而且对整个组件而言也能获得不错的覆盖度。

但是，从设计的角度来说，这种整合测试比不上严格的单元测试和由外而内的 TDD，因为它没有后者的优势全面。

就本章的示例而言，可以比较一下修改前后的代码：

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        if not isinstance(request.user, AnonymousUser):
            list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
        return redirect(list_)
    return render(request, 'home.html', {'form': form})
```

如果想省点儿事，不走隔离测试这条路，你会下功夫重构视图函数吗？我知道写作本书草稿时我不会。我希望自己在真实的项目中会这么做，但也不能保证。可是编写隔离测试却让你看清代码复杂在何处。

23.11.1 以复杂度为准则

不得不说，处理复杂问题时才能体现隔离测试的优势。本书中的例子非常简单，还不太值得这么做。就算是本章的例子，我也能说服自己，完全**不用**编写这些隔离测试。

可一旦应用变得复杂，比如视图和模型之间分了更多层、需要编写辅助方法或自己的类，那多编写一些隔离测试或许就能从中受益了。

23.11.2 两种测试都要写吗

功能测试组件能告诉我们集成各部分代码时是否有问题。隔离测试能帮助我们设计出更好的代码，还能验证细节的处理是否正确。那么中间层集成测试还有其他作用吗？

我想，如果集成测试能更快地提供反馈，或者能更精确地找出集成问题的原因所在，那么答案就是肯定的。集成测试的优势之一是，它在调用跟踪中提供的调试信息比功能测试详细。

甚至还可以把各组件分开——可以编写一个速度快、隔离的单元测试组件，完全不用 manage.py，因为这些测试不需要 Django 测试运行程序提供的任何数据库清理操作。然后使用 Django 提供的工具编写中间层测试，最后使用功能测试检查与过渡服务器交互的各层。如果各层提供的功能循序渐进，或许就可以采用这种方案。

到底怎么做，要根据实际情况而定。我希望读过这一章之后，你能体会到如何权衡。第 26 章会继续讨论这个话题。

23.11.3 继续前行

对新版代码很满意，那就合并到主分支吧：

```
$ git add .
$ git commit -m "add list owners via forms. more isolated tests"
$ git checkout master
$ git checkout -b master-noforms-noisolation-bak # 也可以做个备份
$ git checkout master
$ git reset --hard more-isolation # 把主分支重设到这个分支
```

现在，运行功能测试要花很长时间，我想知道我们能不能做些什么来改善这种状况。

不同测试类型以及解耦 ORM 代码的利弊

功能测试

- 从用户的角度出发，最大程度上保证应用可以正常运行。
- 但是，反馈循环用时长。
- 无法帮助我们写出简洁的代码。

整合测试（依赖于 ORM 或 Django 测试客户端等）

- 编写速度快。
- 易于理解。
- 发现任何集成问题都会提醒你。
- 但是，并不总能得到好的设计（这取决于你自己）。
- 一般运行速度比隔离测试慢。

隔离测试（使用驭件）

- 涉及的工作量最大。
- 可能难以阅读和理解。
- 但是，这种测试最能引导你实现更好的设计。
- 运行速度最快。

解耦应用代码和 ORM 代码

力求隔离测试的后果之一是，我们不得不从视图和表单等处删除 ORM 代码，把它们放到辅助函数或者辅助方法中。如果从解耦应用代码和 ORM 代码的角度看，这么做有好处，还能提高代码的可读性。当然，所有事情都一样，要结合实际情况判断是否值得付出额外精力去做。

第24章

持续集成

网站越变越大，运行所有功能测试的时间也越来越长。如果时长一直增加，我们很可能不再运行功能测试。

为了避免发生这种情况，可以搭建一个“持续集成”（Continuous Integration，简称 CI）服务器，自动运行功能测试。这样，在日常开发中，只需运行当下关注的功能测试，整个测试组件则交给 CI 服务器自动运行。如果不小心破坏了某项功能，CI 服务器会通知我们。单元测试的运行速度一直很快，每隔几秒就可以运行一次。

现在，开发者喜欢使用的 CI 服务器是 Jenkins。Jenkins 使用 Java 开发，经常出问题，界面也不漂亮，但大家都在用，而且插件系统很棒，下面安装并运行 Jenkins。

24.1 安装Jenkins

CI 托管服务有很多，基本上都提供了一个立即就能使用的 Jenkins 服务器。我知道的就有 Sauce Labs、Travis、Circle-CI 和 ShiningPanda，可能还有更多。假设要在自己有控制权的服务器上安装所需的一切软件。

> 把 Jenkins 安装在过渡服务器或生产服务器上可不是个好主意，因为有很多操作要交给 Jenkins 完成，比如重新引导过渡服务器。

要从 Jenkins 的官方 apt 仓库中安装最新版，因为 Ubuntu 默认安装的版本对本地化和 Unicode 支持还有些恼人的问题，而且默认配置也没监听外网：

```
root@server:$ wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key |\
     apt-key add -
root@server:$ echo deb http://pkg.jenkins.io/debian-stable binary/ | tee \
    /etc/apt/sources.list.d/jenkins.list
root@server:$ apt-get update
root@server:$ apt-get install jenkins
```

（这是从 Jenkins 网站上查到的安装说明。）

此外还要安装几个依赖：

```
root@server:$ apt-get install firefox python3-venv xvfb
# 以及构建fabric3所需的依赖
root@server:$ apt-get install build-essential libssl-dev libffi-dev
```

我们还要下载、解压和安装 geckodriver（我写到这里时，版本为 v0.17；你阅读时，记得换成最新版）：

```
root@server:$ wget https://github.com/mozilla/geckodriver/releases\
/download/v0.17.0/geckodriver-v0.17.0-linux64.tar.gz
root@server:$ tar -xvzf geckodriver-v0.17.0-linux64.tar.gz
root@server:$ mv geckodriver /usr/local/bin
root@server:$ geckodriver --version
geckodriver 0.17.0
```

增加一些交换内存

Jenkins 是内存消耗大户。如果在 RAM 很小的虚拟主机上运行，会由于内存不足而崩溃。这时，要增加一些交换内存：

```
$ fallocate -l 4G /swapfile
$ mkswap /swapfile
$ chmod 600 /swapfile
$ swapon /swapfile
```

这样内存就充足了。

24.2 配置Jenkins

现在可以访问服务器的 URL/IP，通过 8080 端口访问 Jenkins，你会看到如图 24-1 所示的界面。

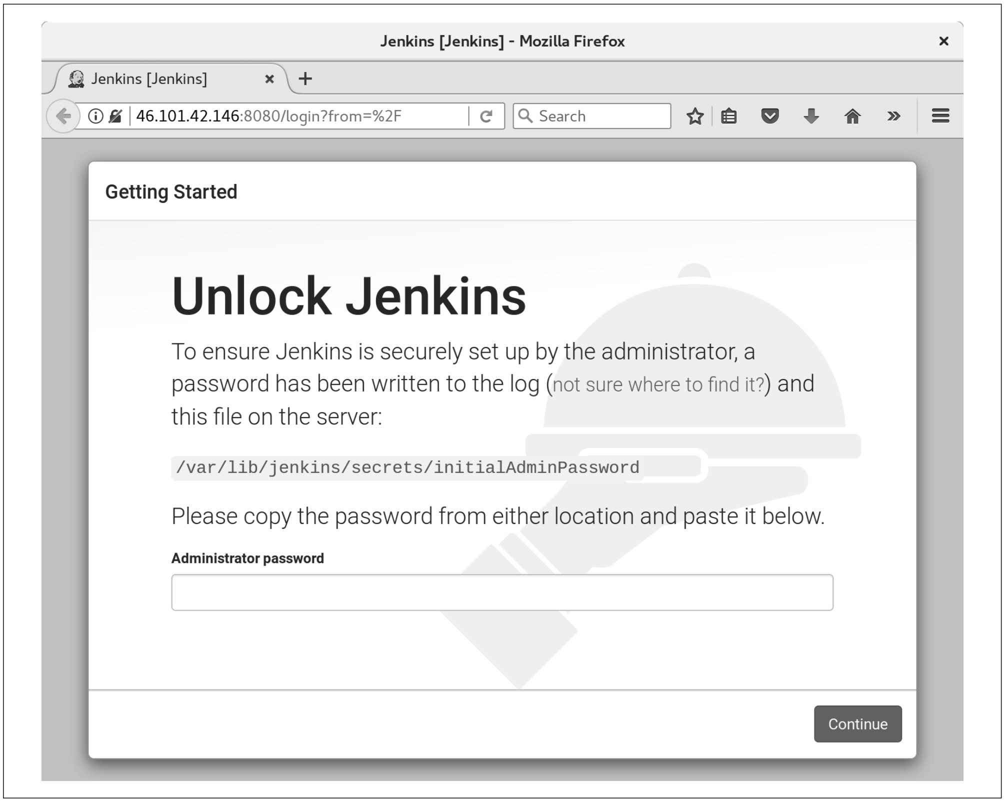


图 24-1：Jenkins 解锁界面

24.2.1 首次解锁

解锁界面告诉我们，首次使用时要从磁盘中读取一个文件，解锁服务器。切换到终端，使用下述命令打印那个文件的内容：

```
root@server$ cat /var/lib/jenkins/secrets/initialAdminPassword
```

24.2.2 现在建议安装的插件

接下来会让你选择安装“推荐的”插件。系统推荐的插件还不错。（作为自尊心强的技术宅，我们会本能地点击“自定义”。一开始我也是这么做的，但是自定义界面也没什么。别担心，稍后会再添加一些插件。）

24.2.3 配置管理员用户

接下来，设置登录 Jenkins 的用户名和密码，如图 24-2 所示。

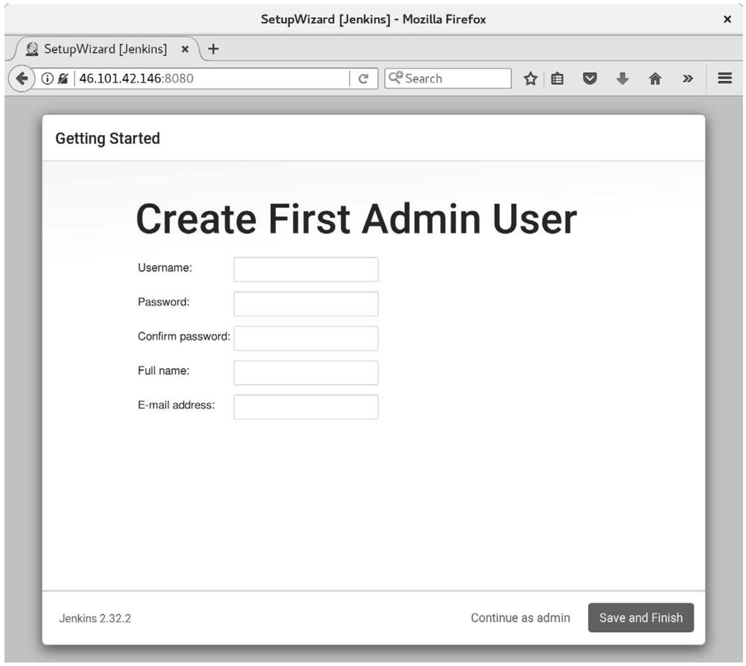


图 24-2：Jenkins 管理员用户配置界面

登录后会看到一个欢迎界面（如图 24-3 所示）。

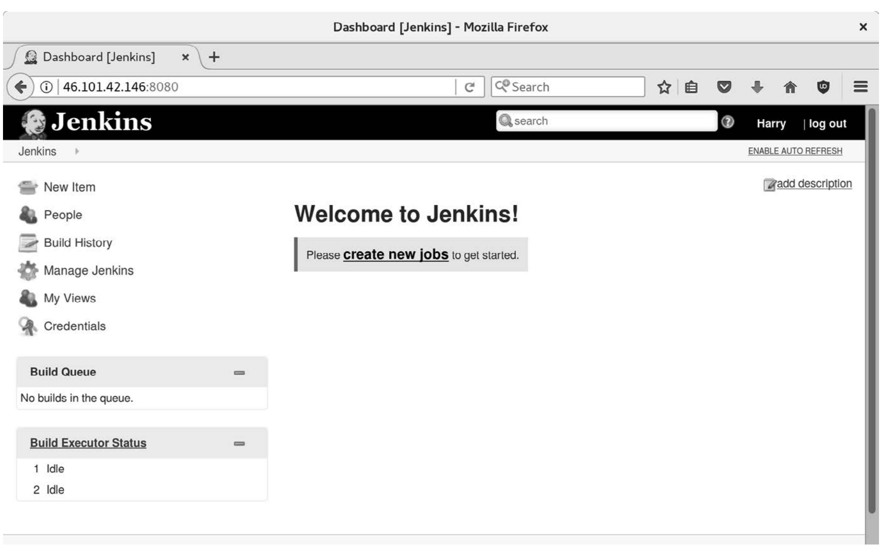


图 24-3：看到一个男仆（左上角），好奇怪

24.2.4　添加插件

依次点击这些链接：Manage Jenkins（管理 Jenkins）→ Manage Plugins（管理插件）→ Available（可用插件）。

我们将安装下述插件：

- ShiningPanda
- Xvfb

点击“Install”（如图 24-4 所示）。

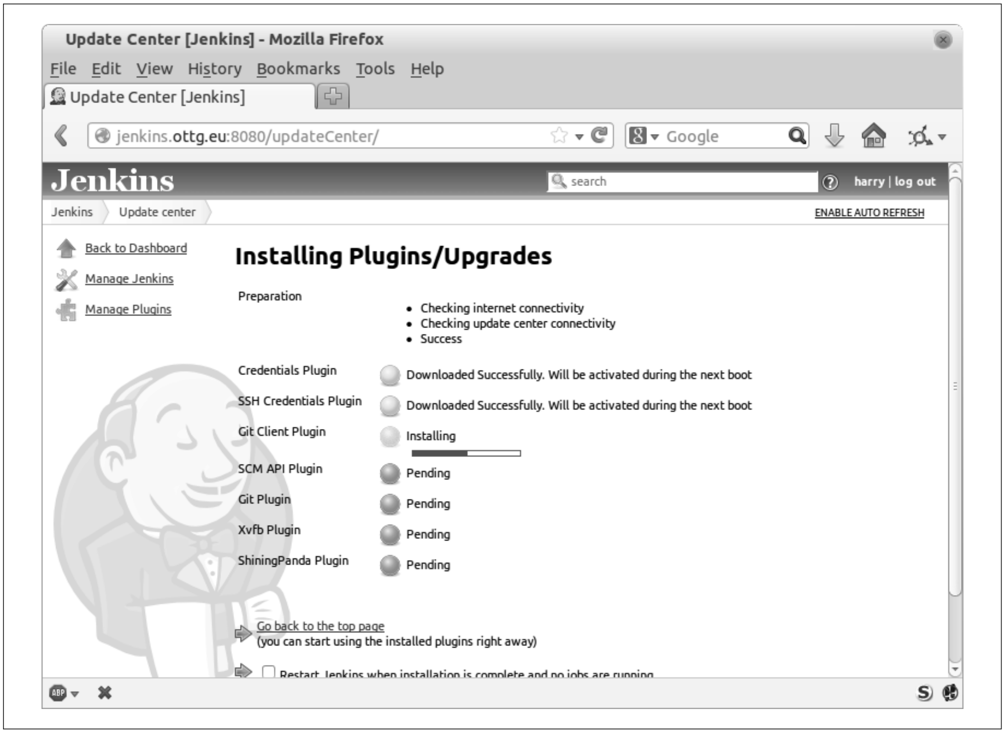


图 24-4：安装插件中……

24.2.5　告诉Jenkins到哪里寻找Python 3和Xvfb

我们要告诉 ShiningPanda 插件，Python 3 安装在哪里（通常是 /usr/bin/python3，不过也可以执行 `which python3` 命令查看）。

- Manage Jenkins（管理 Jenkins）→ Global Tool Configuration（全局工具配置）。
- Python → Python installations（Python 安装位置）→ Add Python（添加 Python，如图 24-5 所示；可以放心忽略提醒消息）。

- Xvfb installation（Xvfb 安装位置）→ Add Xvfb installation（添加 Xvfb 安装位置）；在安装目录中输入 /usr/bin。



图 24-5：Python 安装在哪里

24.2.6　设置HTTPS

为了提升 Jenkins 的安全性，最后还要设置 HTTPS。为此，我们将让 Nginx 使用自签名的证书，把发给 443 端口的请求转发给 8080 端口。这样设置之后，甚至可以让防火墙阻断 8080 端口。具体设置方法这里不详细讨论，你可以参照下述链接给出的说明。

- Jenkins 官方的 Ubuntu 安装指南。
- 如何创建自签名 SSL 证书。
- 如何把 HTTP 重定向到 HTTPS。

24.3　设置项目

现在 Jenkins 基本配置好了，下面设置项目。

- 点击“New Item”按钮。
- 名称输入“Superlists”，选择“Freestyle project”，然后点击“OK”。
- 添加 Git 仓库，如图 24-6 所示。

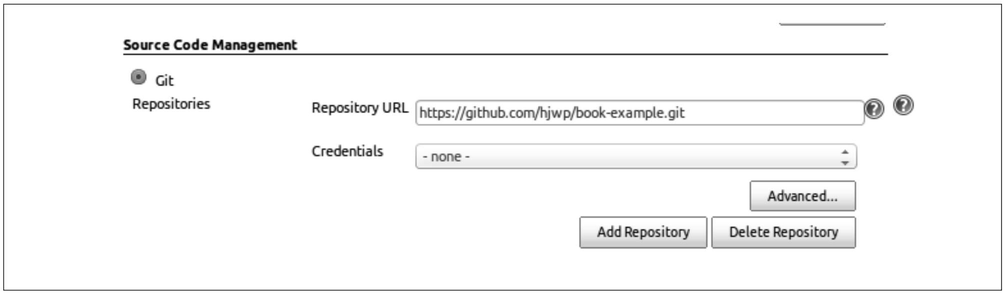


图 24-6：从 Git 仓库中获取源码

- 设为每小时轮询一次（如图 24-7，看一下帮助文本，触发构建操作还有很多其他方式）。

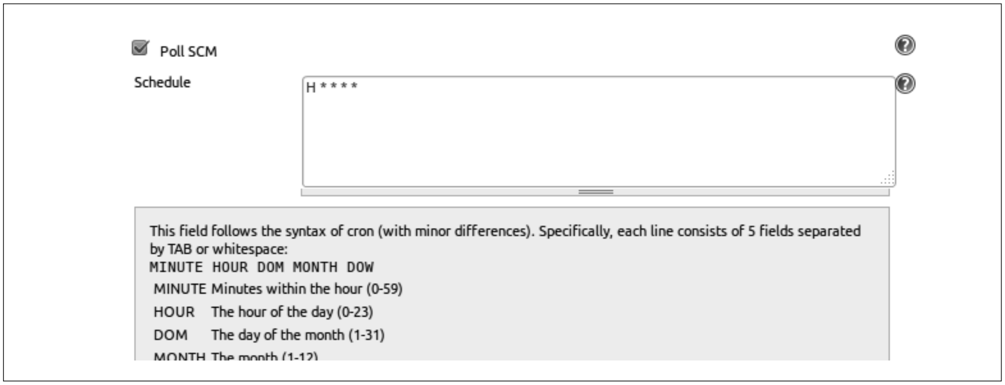


图 24-7：轮询 GitHub，获取改动

- 在一个 Python 3 虚拟环境中运行测试。
- 单元测试和功能测试分开运行，如图 24-8 所示。

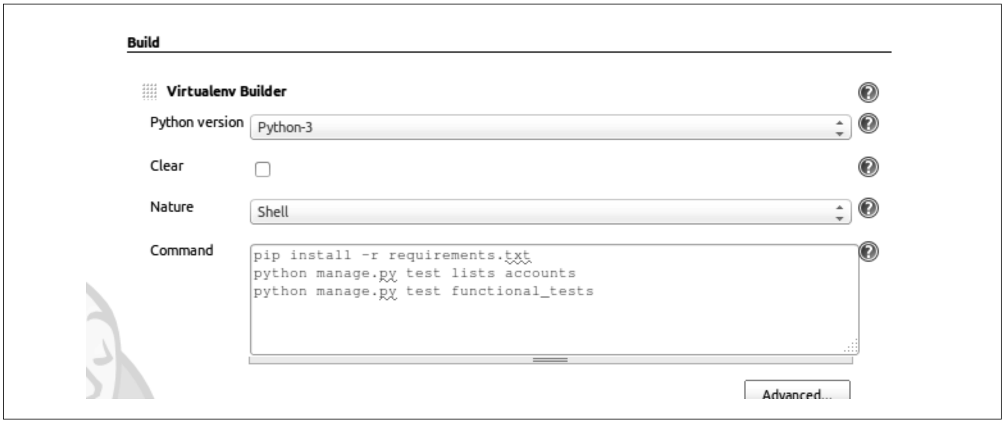


图 24-8：虚拟环境中执行的构建步骤

24.4 第一次构建

点击“Build Now”（现在构建）按钮，然后查看“Console Output”（终端输出），应该会看到类似下面的内容：

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/hjwp/book-example.git
Checking out Revision d515acebf7e173f165ce713b30295a4a6ee17c07 (origin/master)
[workspace] $ /bin/sh -xe /tmp/shiningpanda7260707941304155464.sh
+ pip install -r requirements.txt
Requirement already satisfied (use --upgrade to upgrade): Django==1.11 in
```

```
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.3/
site-packages
(from -r requirements.txt (line 1))

Requirement already satisfied (use --upgrade to upgrade): gunicorn==17.5 in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.3/
site-packages
(from -r requirements.txt (line 3))
Downloading/unpacking requests==2.0.0 (from -r requirements.txt (line 4))
  Running setup.py egg_info for package requests

Installing collected packages: requests
  Running setup.py install for requests

Successfully installed requests
Cleaning up...
+ python manage.py test lists accounts
...................................................................
 ---------------------------------------------------------------------
Ran 67 tests in 0.429s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
+ python manage.py test functional_tests
EEEEEE
======================================================================
ERROR: functional_tests.test_layout_and_styling (unittest.loader._FailedTest)
 ---------------------------------------------------------------------
ImportError: Failed to import test module: functional_tests.test_layout_and_styling
[...]
ImportError: No module named 'selenium'

Ran 6 tests in 0.001s

FAILED (errors=6)

Build step 'Virtualenv Builder' marked build as failure
```

啊，在虚拟环境中要安装 Selenium。

在构建步骤中添加一步，手动安装 Selenium：

```
pip install -r requirements.txt
python manage.py test accounts lists
pip install selenium
python manage.py test functional_tests
```

> 有些人喜欢使用 test-requirements.txt 文件指定测试（不是主应用）需要的包。

然后再次点击“Build Now”（现在构建）按钮。

接下来会发生下面两件事中的一件。你可能会看到控制台输出了类似这样的错误消息：

```
    self.browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: 'The browser appears to
have exited before we could connect. The output was: b"\\n(process:19757):
GLib-CRITICAL **: g_slice_set_config: assertion \'sys_page_size == 0\'
failed\\nError: no display specified\\n"'
[...]
selenium.common.exceptions.WebDriverException: Message: connection refused
```

也可能构建完全挂起（我至少遇到过一次）。出现这种情况的原因是 Firefox 无法启动，因为没有显示器可用。

24.5 设置虚拟显示器，让功能测试能在无界面的环境中运行

从调用跟踪中可以看出，Firefox 无法启动，因为服务器没有显示器。

这个问题有两种解决方法。第一种，换用无界面浏览器（headless browser），例如 PhantomJS 或 SlimerJS。这种工具绝对有存在的意义，最大的特点是运行速度快，但也有缺点。首先，它们不是“真正的”Web 浏览器，所以无法保证能捕获用户使用真正的浏览器时遇到的全部怪异行为。其次，它们在 Selenium 中的表现差异很大，因此要花费大量精力重写功能测试。

> 我只把无界面浏览器当作开发工具，目的是在开发者的设备中提升功能测试的运行速度。在 CI 服务器上运行测试则使用真正的浏览器。

第二种解决方法是设置虚拟显示器：让服务器认为自己连接了显示器，这样 Firefox 就能正常运行了。这种工具很多，我们要使用的是“Xvfb”（X Virtual Framebuffer）[1]，因为它安装和使用都很简单，而且还有一个合用的 Jenkins 插件（现在知道为什么之前要安装它了吧）。

回到项目页面，点击“Configure”（配置），找到“Build Environment”（构建环境）部分。启用虚拟显示器的方法很简单，勾选“Start Xvfb before the build, and shut it down after”（构建前启动 Xvfb，并在构建完成后关闭）即可，如图 24-9 所示。

注 1：如果想在 Python 代码中控制虚拟显示器，可以试试 pyvirtualdisplay。

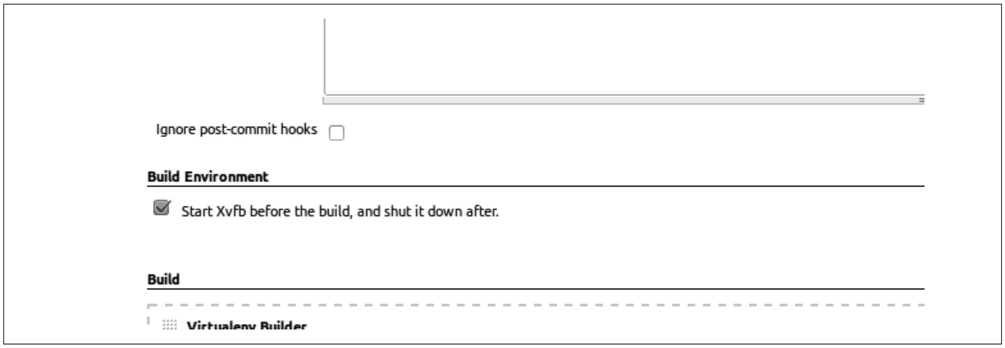


图 24-9：有时配置方式很简单

现在构建过程顺利多了：

```
[...]
Xvfb starting$ /usr/bin/Xvfb :2 -screen 0 1024x768x24 -fbdir
/var/lib/jenkins/2013-11-04_03-27-221510012427739470928xvfb
[...]
+ python manage.py test lists accounts
...............................................................
 ---------------------------------------------------------------------
Ran 63 tests in 0.410s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...

+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.5/
site-packages
Cleaning up...

+ python manage.py test functional_tests
......F.
======================================================================
FAIL: test_can_start_a_list_for_one_user
(functional_tests.test_simple_list_creation.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_simple_list_creation.py", line
43, in test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
  File "/.../superlists/functional_tests/base.py", line 51, in
wait_for_row_in_list_table
    raise e
  File "/.../superlists/functional_tests/base.py", line 47, in
wait_for_row_in_list_table
    self.assertIn(row_text, [row.text for row in rows])
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
 ---------------------------------------------------------------------
```

```
Ran 8 tests in 89.275s

FAILED (errors=1)
Creating test database for alias 'default'...
[{'secure': False, 'domain': 'localhost', 'name': 'sessionid', 'expiry':
1920011311, 'path': '/', 'value': 'a8d8bbde33nreq6gihw8a7r1cc8bf02k'}]
Destroying test database for alias 'default'...
Build step 'Virtualenv Builder' marked build as failure
Xvfb stopping
Finished: FAILURE
```

就快成功了！为了调试错误，还需要截图。

> 这个错误是由于我的 Jenkins 性能不足，所以不一定总会出现。你可能会看到不同的错误，或者根本没错误。不管怎样，下面介绍的截图工具和处理条件竞争的工具总有一天会用到。继续读吧！

24.6 截图

为了调试远程设备中意料之外的失败，最好能看到失败时的屏幕图片，或者还可以转储页面的 HTML。这些操作可在功能测试类中的 `tearDown` 方法里自定义逻辑实现。为此，要深入 `unittest` 的内部，使用私有属性 `_outcomeForDoCleanups`，不过像下面这样写也行：

functional_tests/base.py (ch21l006)

```
import os
from datetime import datetime
[...]

SCREEN_DUMP_LOCATION = os.path.join(
    os.path.dirname(os.path.abspath(__file__)), 'screendumps'
)
[...]

    def tearDown(self):
        if self._test_has_failed():
            if not os.path.exists(SCREEN_DUMP_LOCATION):
                os.makedirs(SCREEN_DUMP_LOCATION)
            for ix, handle in enumerate(self.browser.window_handles):
                self._windowid = ix
                self.browser.switch_to_window(handle)
                self.take_screenshot()
                self.dump_html()
        self.browser.quit()
        super().tearDown()


    def _test_has_failed(self):
       # 有点令人费解，但我找不到更好的方法了
       return any(error for (method, error) in self._outcome.errors)
```

首先，必要时创建存放截图的目录。然后，遍历所有打开的浏览器选项卡和页面，调用一些 Selenium 提供的方法（`get_screenshot_as_file` 和 `browser.page_source`）截图以及转储 HTML：

functional_tests/base.py (ch21l007)

```
    def take_screenshot(self):
        filename = self._get_filename() + '.png'
        print('screenshotting to', filename)
        self.browser.get_screenshot_as_file(filename)


    def dump_html(self):
        filename = self._get_filename() + '.html'
        print('dumping page HTML to', filename)
        with open(filename, 'w') as f:
            f.write(self.browser.page_source)
```

最后，使用一种方式生成唯一的文件名标识符。文件名中包括测试方法和测试类的名字，以及一个时间戳：

functional_tests/base.py (ch21l008)

```
    def _get_filename(self):
        timestamp = datetime.now().isoformat().replace(':', '.')[:19]
        return '{folder}/{classname}.{method}-window{windowid}-{timestamp}'.format(
            folder=SCREEN_DUMP_LOCATION,
            classname=self.__class__.__name__,
            method=self._testMethodName,
            windowid=self._windowid,
            timestamp=timestamp
        )
```

可以先在本地测试一下，故意让某个测试失败，例如使用 `self.fail()`，会看到类似下面的输出：

```
[...]
screenshotting to /.../superlists/functional_tests/screendumps/MyListsTest.test
_logged_in_users_lists_are_saved_as_my_lists-window0-2014-03-09T11.19.12.png
dumping page HTML to /.../superlists/functional_tests/screendumps/MyListsTest.t
est_logged_in_users_lists_are_saved_as_my_lists-window0-[...]
```

删掉 `self.fail()`，然后提交，再推送：

```
$ git diff # 显示base.py中的改动
$ echo "functional_tests/screendumps" >> .gitignore
$ git commit -am "add screenshot on failure to FT runner"
$ git push
```

在 Jenkins 中重新构建时，会看到类似下面的输出：

```
screenshotting to /var/lib/jenkins/jobs/Superlists/.../functional_tests/
screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.png
dumping page HTML to /var/lib/jenkins/jobs/Superlists/.../functional_tests/
screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.html
```

可以在“工作空间”中查看这些文件。工作空间是 Jenkins 用来存储源码以及运行测试所在的文件夹，如图 24-10 所示。

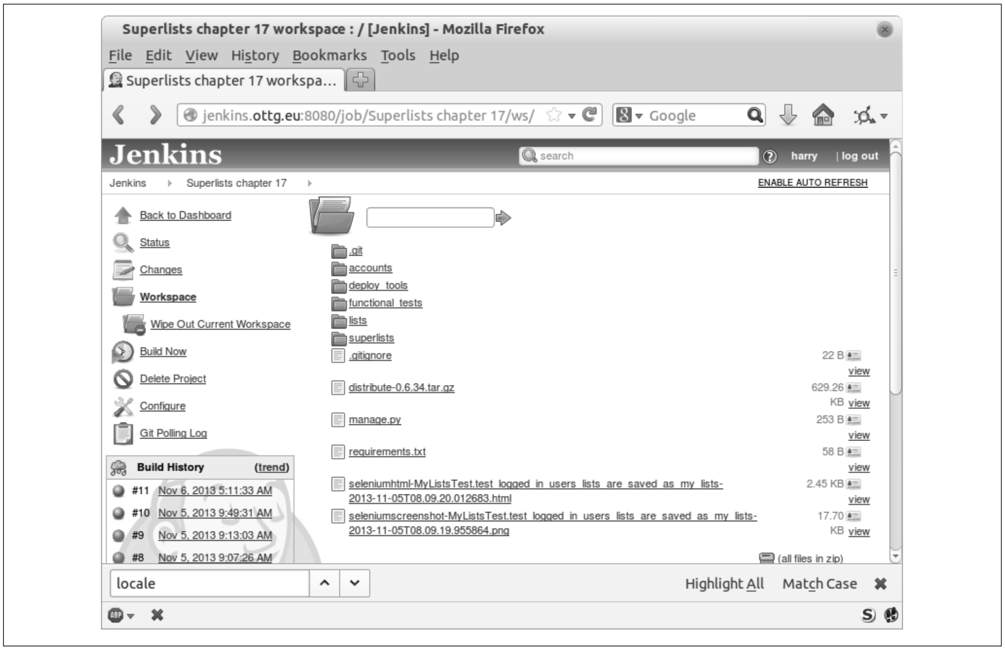


图 24-10：访问项目的工作空间

然后查看截图，如图 24-11 所示。

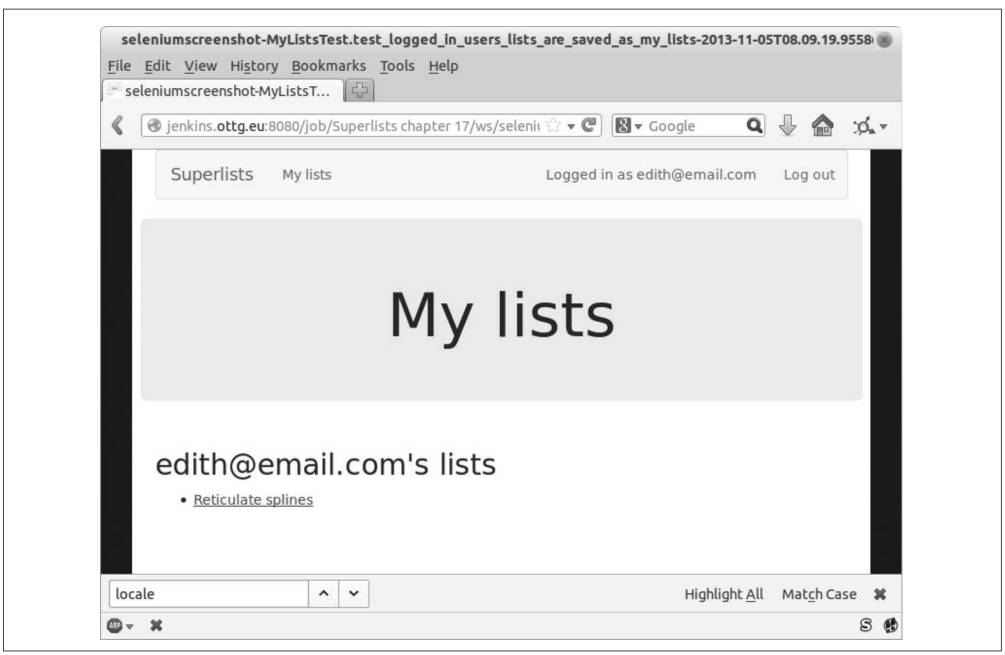


图 24-11：截图看起来正常

24.7　如有疑问，增加超时试试

嗯，显然没什么线索。老话说得好，如有疑问，增加超时：

functional_tests/base.py

```
MAX_WAIT = 20
```

然后在 Jenkins 中点击“Buid now”（现在构建），再次构建，确认测试是否能通过，如图 24-12 所示。

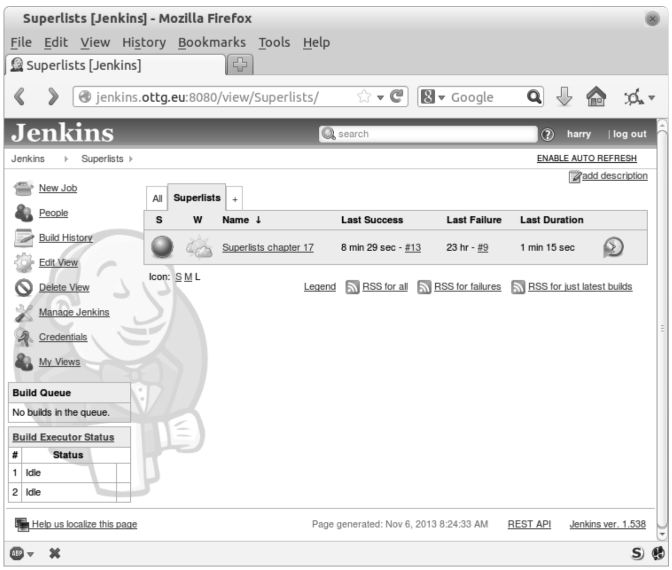


图 24-12：前景更明朗

Jenkins 使用蓝色表示构建成功。居然没用绿色，真让人失望。不过看到太阳从云中探出头来，心情又舒畅了。这个图标表示成功构建和失败构建的平均比值正在发生变化，而且是向好的一面发展。

24.8　使用PhantomJS运行QUnit JavaScript测试

差点儿忘了还有一种测试——JavaScript 测试。现在的“测试运行程序”是真正的 Web 浏览器。若想在 Jenkins 中运行 JavaScript 测试，需要一种命令行测试运行程序。借此机会使用 PhantomJS。

24.8.1　安装node

别再假装用不到 JavaScript 了，做 Web 开发，离不开它。因此，要在自己的电脑中安装 node.js，这一步不可避免。

安装方法参见 node.js 下载页面中的说明。Windows 和 Mac 系统都有安装包，而且各种流行的 Linux 发行版都有各自的包[2]。

安装好 node 之后，可以执行下面的命令安装 PhantomJS：

```
root@server $ npm install -g phantomjs # -g的意思是系统全局安装
```

接下来要下载 QUnit/PhantomJS 测试运行程序。测试运行程序有很多（为了运行本书中的 QUnit 测试，我甚至还自己写过一个简单的），不过最好使用 QUnit 插件页面提到的那个。写作本书时，这个运行程序的仓库地址是 https://github.com/jonkemp/qunit-phantomjs-runner。只需要一个文件，runner.js。

最终得到的文件夹结构如下：

```
$ tree lists/static/tests/
lists/static/tests/
├── qunit-2.0.1.css
├── qunit-2.0.1.js
├── runner.js
└── tests.html

0 directories, 4 files
```

试一下这个运行程序：

```
$ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html
Took 24ms to run 2 tests. 2 passed, 0 failed.
```

保险起见，故意破坏一个测试：

lists/static/list.js (ch21l019)

```
$('input[name="text"]').on('keypress', function () {
  // $('.has-error').hide();
});
```

果然，测试失败了：

```
$ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html

Test failed: errors should be hidden on keypress
    Failed assertion: expected: false, but was: true
file:///.../superlists/lists/static/tests/tests.html:27:15

Took 27ms to run 2 tests. 1 passed, 1 failed.
```

很好！再改回去，提交并推送运行程序，然后将其添加到 Jenkins 的构建步骤中：

```
$ git checkout lists/static/list.js
$ git add lists/static/tests/runner.js
$ git commit -m "Add phantomjs test runner for javascript tests"
$ git push
```

注 2：一定要下载最新版。在 Ubuntu 中别用默认的包，要使用 PPA。

24.8.2　在Jenkins中添加构建步骤

再次编辑项目配置，为每个 JavaScript 测试文件添加一个构建步骤，如图 24-13 所示。

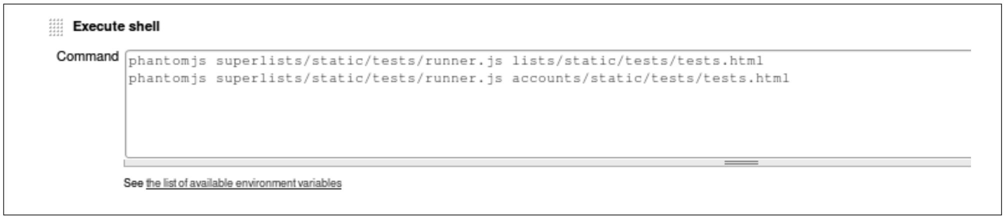


图 24-13：为 JavaScript 单元测试添加构建步骤

还要在服务器中安装 PhantomJS：

```
root@server:$ add-apt-repository -y ppa:chris-lea/node.js
root@server:$ apt-get update
root@server:$ apt-get install nodejs
root@server:$ npm install -g phantomjs
```

至此，编写了完整的 CI 构建步骤，能运行所有测试！

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/hjwp/book-example.git
Checking out Revision 936a484038194b289312ff62f10d24e6a054fb29 (origin/chapter_1
Xvfb starting$ /usr/bin/Xvfb :1 -screen 0 1024x768x24 -fbdir /var/lib/jenkins/20
[workspace] $ /bin/sh -xe /tmp/shiningpanda7092102504259037999.sh

+ pip install -r requirements.txt
[...]

+ python manage.py test lists
.................................
 ---------------------------------------------------------------------
Ran 43 tests in 0.229s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...

+ python manage.py test accounts
..................
 ---------------------------------------------------------------------
Ran 18 tests in 0.078s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...

[workspace] $ /bin/sh -xe /tmp/hudson2967478575201471277.sh
+ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html
Took 32ms to run 2 tests. 2 passed, 0 failed.
+ phantomjs lists/static/tests/runner.js accounts/static/tests/tests.html
```

```
Took 47ms to run 11 tests. 11 passed, 0 failed.

[workspace] $ /bin/sh -xe /tmp/shiningpanda7526089957247195819.sh
+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in /var/lib/

Cleaning up...
[workspace] $ /bin/sh -xe /tmp/shiningpanda2420240268202055029.sh
+ python manage.py test functional_tests
........
 ----------------------------------------------------------------------
Ran 8 tests in 76.804s

OK
```

如果我太懒，不想在自己的设备中运行整个测试组件，CI 服务器可以代劳——真是太好了。测试山羊的另一个代理人正在网络空间里监视我们呢！

24.9 CI服务器能完成的其他操作

Jenkins 和 CI 服务器的作用只介绍了皮毛。例如，还可以让 CI 服务器在监控仓库的新提交方面变得更智能。

或者做些更有趣的事，除了运行普通的功能测试之外，还可以使用 CI 服务器自动运行过渡服务器中的测试。如果所有功能测试都能通过，你可以添加一个构建步骤，把代码部署到过渡服务器中，然后在过渡服务器中再运行功能测试。这样整个过程又多了一步可以自动完成，而且可以保证过渡服务器始终使用最新的代码。

有些人甚至使用 CI 服务器把最新发布的代码部署到生产服务器中。

CI 和 Selenium 最佳实践

- 尽早为自己的项目搭建 CI 服务器
  一旦运行功能测试所花的时间超过几秒钟，你就会发现自己根本不想再运行了。把这个任务交给 CI 服务器吧，确保所有测试都能在某处运行。
- 测试失败时截图和转储 HTML
  如果你能看到测试失败时网页是什么样，调试就容易得多。截图和转储 HTML 有助于调试 CI 服务器中的失败，而且对本地运行的测试也很有用。
- 时刻准备调整超时
  CI 服务器的运行速度可能没有你的笔记本电脑快，尤其是同时运行多个测试、负载较高时。你要时刻准备调整超时，把它设为较大的值，尽量降低随机失败的概率。
- 想办法把 CI 和过渡服务器连接起来
  使用 `LiveServerTestCase` 的测试在开发环境中不会遇到什么问题，但若想得到十足的保障，就要在真正的服务器中运行测试。想办法让 CI 服务器把代码部署到过渡服务器中，然后在过渡服务器中运行功能测试。这么做还有个附带好处：测试自动化部署脚本是否可用。

第25章

简单的社会化功能、页面模式以及练习

“现在一切都要社会化”的调侃是不是有点过时了？不管怎样，现在一切都是经过 A/B 测试和大数据分析能得到超多点击量的列表，类似“创意导师认为将颠覆你观念的十件事”。不管是否真能激发创意，列表都更容易传播。那我们就让用户能和其他人协作完成他们的列表吧。

在实现这个功能的过程中，我们将使用页面对象模式（Page Object pattern）改进功能测试。

我不告诉你具体做法，而是让你自己编写单元测试和应用代码。别担心，我不会让你完全自己动手，会告诉你大概步骤和一些提示。

25.1　有多个用户以及使用addCleanup的功能测试

开始吧。这个功能测试需要两个用户：

functional_tests/test_sharing.py (ch22l001)

```
from selenium import webdriver
from .base import FunctionalTest


def quit_if_possible(browser):
    try: browser.quit()
    except: pass
```

```
class SharingTest(FunctionalTest):

    def test_can_share_a_list_with_another_user(self):
        # 伊迪丝是已登录用户
        self.create_pre_authenticated_session('edith@example.com')
        edith_browser = self.browser
        self.addCleanup(lambda: quit_if_possible(edith_browser))

        # 她的朋友Oniciferous也在使用这个清单网站
        oni_browser = webdriver.Firefox()
        self.addCleanup(lambda: quit_if_possible(oni_browser))
        self.browser = oni_browser
        self.create_pre_authenticated_session('oniciferous@example.com')

        # 伊迪丝访问首页，新建一个清单
        self.browser = edith_browser
        self.browser.get(self.live_server_url)
        self.add_list_item('Get help')

        # 她看到“分享这个清单”选项
        share_box = self.browser.find_element_by_css_selector(
            'input[name="sharee"]'
        )
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )
```

这一节有个功能值得注意：addCleanup 函数，它的文档可以在线查看。这个函数可以代替 tearDown 函数，清理测试中使用的资源。如果资源在测试运行的过程中才用到，最好使用 addCleanup 函数，因为这样就不用在 tearDown 函数中花时间区分哪些资源需要清理，哪些不需要清理。

addCleanup 函数在 tearDown 函数之后运行，所以在 quit_if_possible 函数中才要使用 try/except 语句，因为不管 edith_browser 和 oni_browser 中哪一个的值是 self.browser，测试结束时 tearDown 函数都会关闭这个浏览器。

还要把测试方法 create_pre_authenticated_session 从 test_my_lists.py 移到 base.py 中。

好了，看一下测试结果如何：

```
$ python manage.py test functional_tests.test_sharing
[...]
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_sharing.py", line 31, in
test_can_share_a_list_with_another_user
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: input[name="sharee"]
```

太好了，看样子可以创建两个用户会话，而且得到了一个意料之中的失败，因为页面中没有填写电子邮件地址的输入框，无法分享给别人。

现在做一次提交，因为至少已经编写了一个占位功能测试，也移动了 create_pre_authenticated_session 函数，接下来要重构功能测试：

```
$ git add functional_tests
$ git commit -m "New FT for sharing, move session creation stuff to base"
```

25.2 页面模式

在继续之前，我想再展示一种减少功能测试中重复代码的方式，叫作“页面对象”。

我们为功能测试构建了几个辅助方法，例如这里使用的 add_list_item。但如果不断构建辅助方法，测试也会变得臃肿不堪。我曾处理过一个超过 1500 行的功能测试基类，想改一下都十分困难。

此时便可以使用页面对象，尽量把网站中不同类型页面的所有信息和辅助方法放在一处。下面来看如何在网站中使用页面对象。首先是一个表示清单页面的类：

functional_tests/list_page.py

```
from selenium.webdriver.common.keys import Keys
from .base import wait


class ListPage(object):

    def __init__(self, test):
        self.test = test   ❶


    def get_table_rows(self):   ❸
        return self.test.browser.find_elements_by_css_selector('#id_list_table tr')


    @wait
    def wait_for_row_in_list_table(self, item_text, item_number):   ❷
        expected_row_text = f'{item_number}: {item_text}'
        rows = self.get_table_rows()
        self.test.assertIn(expected_row_text, [row.text for row in rows])


    def get_item_input_box(self):   ❷
        return self.test.browser.find_element_by_id('id_text')


    def add_list_item(self, item_text):   ❷
        new_item_no = len(self.get_table_rows()) + 1
        self.get_item_input_box().send_keys(item_text)
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table(item_text, new_item_no)
        return self   ❹
```

❶ 使用表示当前测试的对象初始化，这样就能声明断言、通过 self.test.browser 访问浏览器实例，以及使用 self.test.wait_for 函数。

❷ 从 base.py 中复制一些现有的辅助方法过来，不过稍微做了一点调整……

❸ 例如，使用了这个新方法。

❹ 返回 `self` 只是一种便利措施，以便串接方法（稍后就会用到）。

下面看一下如何在测试中使用页面对象：

functional_tests/test_sharing.py (ch22l004)

```
from .list_page import ListPage
[...]

        # 伊迪丝访问首页，新建一个清单
        self.browser = edith_browser
        list_page = ListPage(self).add_list_item('Get help')
```

继续改写测试，只要想访问列表页面中的元素，就使用页面对象：

functional_tests/test_sharing.py (ch22l008)

```
        # 她看到“分享这个清单”选项
        share_box = list_page.get_share_box()
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )

        # 她分享自己的清单之后，页面更新了
        # 提示已经分享给Oniciferous
        list_page.share_list_with('oniciferous@example.com')
```

我们要在 `ListPage` 类中添加以下三个方法：

functional_tests/list_page.py (ch22l009)

```
    def get_share_box(self):
        return self.test.browser.find_element_by_css_selector(
            'input[name="sharee"]'
        )


    def get_shared_with_list(self):
        return self.test.browser.find_elements_by_css_selector(
            '.list-sharee'
        )


    def share_list_with(self, email):
        self.get_share_box().send_keys(email)
        self.get_share_box().send_keys(Keys.ENTER)
        self.test.wait_for(lambda: self.test.assertIn(
            email,
            [item.text for item in self.get_shared_with_list()]
        ))
```

页面模型背后的思想是，把网站中某个页面的所有信息都集中放在一个地方，如果以后想要修改这个页面，比如简单的调整 HTML 布局，功能测试只需改动一个地方，不用到处修改多个功能测试。

接下来要继续重构其他功能测试。在这里我就不细说了，你可以试着自己完成，感受一下在 DRY 原则和测试可读性方面要做哪些折中处理。

25.3　扩展功能测试测试第二个用户和“My Lists”页面

把分享功能的用户故事写得更详细点儿。伊迪丝在她的清单页面看到这个清单已经分享给 Oniciferous，然后 Oniciferous 登录，看到这个清单出现在“My Lists”页面中，或许显示在“分享给我的清单”中：

functional_tests/test_sharing.py (ch22l010)

```
from .my_lists_page import MyListsPage
[...]

        list_page.share_list_with('oniciferous@example.com')

        # 现在Oniciferous在他的浏览器中访问清单页面
        self.browser = oni_browser
        MyListsPage(self).go_to_my_lists_page()

        # 他看到了伊迪丝分享的清单
        self.browser.find_element_by_link_text('Get help').click()
```

为此，要在 `MyListPage` 类中再定义一个方法：

functional_tests/my_lists_page.py (ch22l011)

```
class MyListsPage(object):

    def __init__(self, test):
        self.test = test


    def go_to_my_lists_page(self):
        self.test.browser.get(self.test.live_server_url)
        self.test.browser.find_element_by_link_text('My lists').click()
        self.test.wait_for(lambda: self.test.assertEqual(
            self.test.browser.find_element_by_tag_name('h1').text,
            'My Lists'
        ))
        return self
```

这个方法最好放到 test_my_lists.py 中，或许还可以再定义一个 `MyListsPage` 类。

现在，Oniciferous 也可以在这个清单中添加待办事项：

functional_tests/test_sharing.py (ch22l012)

```
        # 在清单页面，Oniciferous看到这个清单属于伊迪丝
        self.wait_for(lambda: self.assertEqual(
            list_page.get_list_owner(),
            'edith@example.com'
        ))

        # 他在这个清单中添加一个待办事项
        list_page.add_list_item('Hi 伊迪丝!')

        # 伊迪丝刷新页面后，看到Oniciferous添加的内容
        self.browser = edith_browser
        self.browser.refresh()
        list_page.wait_for_row_in_list_table('Hi Edith!', 2)
```

为此，要在 ListPage 类中添加一个方法：

functional_tests/list_page.py (ch22l013)

```
class ListPage(object):
    [...]

    def get_list_owner(self):
        return self.test.browser.find_element_by_id('id_list_owner').text
```

早就该运行功能测试了，看看这些测试能否通过：

```
$ python manage.py test functional_tests.test_sharing

    share_box = list_page.get_share_box()
    [...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: input[name="sharee"]
```

这个失败在预料之中，因为还没在页面中添加输入框，填写电子邮件地址，分享给别人。做次提交：

```
$ git add functional_tests
$ git commit -m "Create Page objects for list pages, use in sharing FT"
```

25.4 留给读者的练习

> 做完 25.4 节的练习之后，我才算完全明白自己在做什么。
>
> ——Iain H.（读者）

若想牢固掌握所学，没什么比得上自己动手实践。所以，我希望你能试着去做下述练习。

大致步骤如下。

(1) 在 list.html 添加一个新区域，先写一个表单，表单中包含一个输入框，用来输入电子邮件地址。功能测试应该会前进一步。

(2) 需要一个视图，处理表单。先在模板中定义 URL，例如 lists/<list_id>/share。

(3) 然后，编写第一个单元测试，驱动我们定义占位视图。我们希望这个视图处理 POST 请求，响应是重定向，指向清单页面，所以这个测试可以命名为 `ShareListTest.test_post_redirects_to_lists_page`。

(4) 编写占位视图，只需两行代码，一行用于查找清单，一行用于重定向。

(5) 可以再编写一个单元测试，在测试中创建一个用户和一个清单，在 POST 请求中发送电子邮件地址，然后检查 `list_.shared_with.all()`（类似于“My Lists”页面使用的那个 ORM 用法）中是否包含这个用户。`shared_with` 属性还不存在，我们使用的是由外而内的方式。

(6) 所以在这个测试通过之前，要下移到模型层。下一个测试要写入 test_models.py 中。在这个测试中，可以检查清单能否响应 `shared_with.add` 方法。这个方法的参数是用户的电子邮件地址。然后检查清单的 `shared_with.all()` 查询集合中是否包含这个用户。

(7) 然后需要用到 `ManyToManyField`。或许你会看到一条错误消息，提示 `related_name` 有冲突，查阅 Django 的文档之后你会找到解决办法。

(8) 需要执行一次数据库迁移。

(9) 然后，模型测试应该可以通过。回过头来修正视图测试。

(10) 可能会发现重定向视图的测试失败，因为视图发送的 POST 请求无效。可以选择忽略无效的输入，也可以调整测试，发送有效的 POST 请求。

(11) 然后回到模板层。“My Lists”页面需要一个 `<ul>` 元素，使用 `for` 循环列出分享给这个用户的清单。我们还想在清单页面显示这个清单分享给谁了，并注明这个清单的属主是谁。各元素的类和 ID 参见功能测试。如果需要，还可以为这几个需求编写简单的单元测试。

(12) 执行 `runserver` 命令让网站运行起来，或许能帮助你解决问题，以及调整布局和外观。如果使用隐私浏览器会话，可以同时登录多个用户。

最终，可能会得到类似图 25-1 所示的页面。

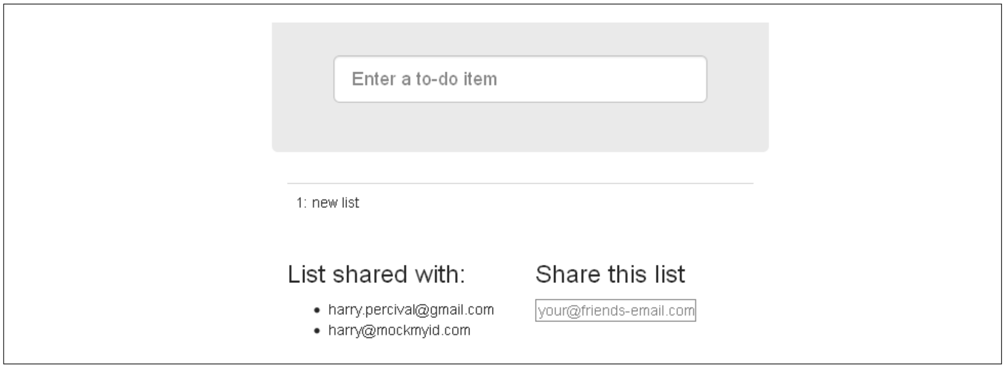


图 25-1：分享清单

页面模式以及真正留给读者的练习

- 在功能测试中运用 DRY 原则
  功能测试多起来后，就会发现不同的测试使用了 UI 的同一部分。尽量避免在多个功能测试中使用重复的常量，例如某个 UI 元素 HTML 代码中的 ID 和类。
- 页面模式
  把辅助方法移到 `FunctionalTest` 基类中会把这个类变得臃肿不堪。可以考虑把处理网站特定部分的全部逻辑保存到单独的页面对象中。
- 留给读者的练习
  希望你真的会做这个练习！试着遵守由外而内的开发方式，如果卡住了，偶尔也可以手动测试。当然，真正留给读者的练习是，在你的下一个项目中使用 TDD。希望它能给你带来愉悦的体验！

下一章做个总结，探讨测试的“最佳实践”。

第 26 章

测试运行速度的快慢和炽热的岩浆

“数据库是炽热的岩浆！”

—— Casey Kinsey

在第 23 章之前，书中几乎所有的“单元”测试或许都应该叫作**整合**测试，因为这些测试要不依赖于数据库，要不使用 Django 测试客户端，请求、响应和视图函数之间的中间层有太多细节都被隐藏了。

有种说法认为，真正的单元测试一定要隔离，因为单元测试只应该测试软件单独的一部分。如果涉及数据库，那就不是单元测试。数据库是炽热的岩浆！

一些 TDD 老手说，你应该尽力编写完全隔离的单元测试，而不要编写整合测试。测试社区一直都有这样的争论，有时还很白热化。

我只是个狂妄的年轻人，对这场争论的细节并不太了解。但在这一章里，我想试着分析人们为什么如此在意这件事，然后给出一些建议，告诉你什么时候勉强可以使用整合测试（我承认很多时候我都是这样做的），什么时候值得争取编写更纯粹的单元测试。

术语：测试的不同类型

- 隔离测试（纯粹的单元测试）与整合测试

  单元测试的主要作用应该是验证应用的逻辑是否正确。**隔离**测试只能测试一部分代码，测试是否通过与其他任何外部代码都没有关系。我所说的纯粹的单元测试是指，对一个函数的测试而言，只有这个函数能让测试失败。如果这个函数依赖于其他系统且破坏这个系统会导致测试失败，就说明这是整合测试。这个系统可以是外部系统，例如数据库，也可以是我们无法控制的另一个函数。不管怎样，只要破坏系统会导致测试失败，这个测试就没有完全隔离，因此也就不是纯粹的单元测试。整合测试并非不好，只不过可能意味着同时测试两个功能。

- 集成测试

  集成测试用于检查被你控制的代码是否能和你无法控制的外部系统完好集成。**集成**测试往往也是**整合**测试。

- 系统测试

  如果说集成测试检查的是与外部系统的集成情况，那么系统测试就是检查应用内部多个系统之间的集成情况。例如，检查数据库、静态文件和服务器配置在一起是否能正常运行。

- 功能测试和验收测试

  验收测试的作用是从用户的角度检查系统是否能正常运行。（用户能接受这种行为吗？）验收测试很难不写成全栈端到端测试。在前文中，使用功能测试代替验收测试和系统测试。

请原谅我的自命不凡，下面我要使用一些哲学术语，以黑格尔辩证法的结构讨论这些问题。

- 正题：纯粹的单元测试运行速度快。
- 反题：编写纯粹的单元测试有哪些风险？
- 合题：讨论一些最佳实践，例如“端口和适配器”“函数式核心，命令式外壳”以及我们到底想从测试中得到什么。

26.1 正题：单元测试除了运行速度超快之外还有其他优势

关于单元测试你经常会听到一种说法：单元测试运行速度快多了。其实，我不觉得这是单元测试的主要优势，不过速度的确值得一谈。

26.1.1 测试运行得越快，开发速度越快

在其他条件相同的情况下，单元测试运行的速度越快越好。可以适当推理出，所有测试运行的速度**都是**越快越好。

本书前文已经概括了 TDD 测试 / 编写代码循环。你已经开始习惯 TDD 流程，时而编写最少量的代码，时而运行测试。以后，一分钟内你要多次运行单元测试，一天之内要多次运行功能测试。

所以，简单而言，测试运行的时间越长，等待测试运行完毕的时间就越长，因此也就拖慢了开发进度。而且问题还不止于此。

26.1.2 神赐的心流状态

现在从社会学角度分析。我们程序员有自己的文化，有自己的族群信仰。这个族群分成很多群体，例如崇拜 TDD 的群体（你现在已经成为其中一员）。有些人喜欢 vi，还有些人离

经叛道，喜欢 emacs。但我们都认同一件事：神赐的心流状态——这是一种精神上的练习，我们自己的冥想方式。我们的精神完全专注，几个小时弹指一挥间就过去，代码自然而然地从指间流出，问题虽然乏味棘手，但难不倒我们。

如果花时间等待慢吞吞的测试组件运行完毕，肯定无法进入心流状态。只要超过几秒钟，你的注意力就会分散，环境也会变化，导致心流状态消失。心流状态就像梦境一样，只要消失，至少要花 15 分钟才能重现。

26.1.3　经常不想运行速度慢的测试，导致代码变坏

如果测试组件运行得慢，你会失去耐心，不想运行测试，这会导致问题横行。我们也许会羞于重构代码，因为知道重构后要花很多时间等待所有测试运行完毕。这两种情况都会导致代码变坏。

26.1.4　现在还行，不过随着时间推移，整合测试会变得越来越慢

你可能觉得没事，测试组件中有很多整合测试，超过 50 个，但运行只用了 0.2 秒。

可是要知道，这个应用很简单。一旦应用变得复杂，数据库中的表和列越来越多，整合测试就会变得越来越慢。在两个测试之间让 Django 重建数据库所用的时间会越来越长。

26.1.5　别只听我一个人说

Gary Bernhardt 的测试经验比我丰富，他在演讲“Fast Test, Slow Test”中生动地阐述了这些观点。推荐你看一下演讲视频。

26.1.6　单元测试能驱使我们实现好的设计

但是，比上述几点更重要的好处或许我在第 23 章已经说过。为了编写隔离性好的单元测试，必须知道依赖下一层中的什么功能，而且要使用整合测试无法实现的解耦式架构，这有助于设计出更好的代码。

26.2　纯粹的单元测试有什么问题

说完优点我们要来个大转折。编写隔离的单元测试也有其危害，尤其是对于我们（包括我和你）这些 TDD 新手而言。

26.2.1　隔离的测试难读也难写

回忆一下我第一个隔离的单元测试，是不是很丑？我承认，重构时把代码移到表单中有些改进，但想一下如果没这么做呢？代码基中就会有一个十分难读的测试。就算是这个测试的最终版本，也仍有一些比较难理解的部分。

26.2.2 隔离测试不会自动测试集成情况

稍后我们会得知，隔离测试只测试当前关注的单元，而且是在隔离的环境中测试，这种测试本性如此，不测试各单元之间的集成情况。

这个问题众所周知，也有很多缓解的方法。不过前文已经说过，这些缓解措施对程序员来说意味着要付出很多艰苦努力：程序员要记住各单元的界面，要分清每个组件需要履行的合约，除了要为单元的内部功能编写测试之外，还得为合约编写测试。

26.2.3 单元测试几乎不能捕获意料之外的问题

单元测试能帮助你捕获差一错误和逻辑混乱导致的错误，这些错误在编写代码时经常会出现，我们知道这一点，所以这些错误在意料之中。不过出现预料之外的问题时，单元测试不会提醒你。如果忘记创建数据库迁移，单元测试不会提醒你；如果中间层自作聪明转义了 HTML 实体，从而影响数据的渲染方式，显示成“唐纳德·拉姆斯菲尔德的 XX”，单元测试也不会提醒你。

26.2.4 使用驭件的测试可能和实现方式联系紧密

最后还有个问题，使用驭件的测试可能和实现方式之间过度耦合。如果你选择使用 `List.objects.create()` 创建对象，但是驭件希望你使用 `List()` 和 `.save()`，这时就算两种用法的实际效果一样，测试也会失败。如果不小心，还可能导致测试本该具有的一个好处缺失，即鼓励重构。如果想修改一个内部 API，你会发现自己要修改很多使用驭件的测试和合约测试。

注意，处理你无法控制的 API 时，这可能不单是一个问题那么简单。你可能还记得我们如何拐弯抹角地测试表单：创建两个 Django 模型驭件，然后使用 `side_effect` 检查环境的状态。如果编写的代码完全在自己的控制之中，你可能想设计自己的内部 API，这样写出的代码更简洁，而且测试时不用拐这么多弯。

26.2.5 这些问题都可以解决

但是，倡导编写隔离测试的人会过来告诉你，这些问题都可以缓解，你要熟练编写隔离测试，还得进入神赐的心流状态。

现在我们论证到哪一步了？

26.3 合题：我们到底想从测试中得到什么

退一步想一下，我们想从测试中得到什么好处，为什么一开始要编写测试？

26.3.1 正确性

我们希望应用没有问题，不管是差一错误之类的低层逻辑错误，还是高层问题，例如软件最终应该提供用户所需的功能。我们想知道是否引入了回归，导致以前能用的功能失效，

而且想在用户察觉之前发现。我们还期望测试告诉我们应用可以正常运行。

26.3.2 简洁可维护的代码

我们希望代码遵守 YAGNI 和 DRY 等原则；希望代码清晰地表明意图，使用合理的方式分成多个组件，而且各组件作用明确、容易理解；希望从测试中获取自信，可以放心地不断重构应用，这样才不会害怕尝试改进设计；还希望测试能主动帮我们找到正确的设计。

26.3.3 高效的工作流程

最后，我们希望测试能帮助实现一种快速高效的工作流程；希望测试有助于减轻开发压力，而且避免让我们犯一些愚蠢的错误；希望测试能让我们始终处于心流状态，因为心流状态不仅令人享受，而且助人提高工作效率；希望测试尽快对我们的工作做出反馈，这样就能尝试新想法，并尽早改进。而且，改进代码时，如果测试不能提供帮助，我们也不想让它成为障碍。

26.3.4 根据所需的优势评估测试

我觉得应该编写多少测试，以及功能测试、整合测试和隔离测试的量怎么分配，没有通用的规则，因为每个项目的情况不同。但可以把所有测试都纳入考虑范围（如表 26-1 所示），然后考量各种测试，看它们能否提供你需要的优势，由此做出判断。

表26-1：不同类型的测试如何帮助我们达成目标

目标	一些考量事项
正确性	• 站在用户的角度看，功能测试的数量是否足够保证应用**真的**能正常运行？ • 各种边界情况彻底测试了吗？感觉这是低层隔离测试的任务。 • 有没有编写测试检查所有组件之间是否能正确配合？要不要编写一些整合测试，或者只用功能测试就行？
简洁可维护的代码	• 测试有没有给我重构代码的自信，而且可以无所畏惧地频繁重构？ • 测试有没有帮我得到一个好的设计？如果整合测试较多、隔离测试较少，我要投入精力为应用的哪一部分编写更多隔离测试，才能得到关于设计更全面的反馈？
高效的工作流程	• 反馈循环的速度令我满意吗？我什么时候能得到问题的提醒？有没有某种方法可以让提醒更早出现？ • 如果高层功能测试很多，运行时间很长，要花整晚时间才能得到意外回归的反馈，有没有一种方法可以让我编写速度更快的测试，整合测试也行，让我早点儿得到反馈？ • 如果需要，我能否运行整个测试组件的一个子集？ • 我是否花了太多时间等待测试运行完毕，导致高效率的心流状态时间缩短？

26.4 架构方案

还有一些架构方案可以帮助测试组件发挥最大的作用，而且特别有助于避免隔离测试的缺点。

这些架构方案大都要求找到系统的边缘，即代码和外部系统（例如数据库、文件系统、万维网或者 UI）交互的地方，然后尝试将外部系统和应用的核心业务逻辑区分开。

26.4.1 端口和适配器（或六边形、简洁）架构

集成测试在系统的**边界**，也就是代码和外部系统（例如数据库、文件系统或 UI 组件）集成的地方，作用最大。

所以，也就是在边界，隔离测试和驭件的作用最小，因为在边界如果测试和某种实现方式耦合过于紧密，最有可能干扰你，或者在边界需要进一步确认各组件之间是否正确集成。

相反，应用的**核心**代码（只关注业务逻辑和业务规则的代码，完全在我们控制之中的代码）不太需要整合测试，因为我们能控制也能理解这些代码。

所以，实现需求的一种方法是尽量减少处理边界的代码量。这样，就可以使用隔离测试检查核心业务逻辑，使用整合测试检查集成点。

Steve Freeman 和 Nat Pryce 在他们合著的书 *Growing Object-Oriented Software*，*Guided by Tests* 中把这种方案称为“端口和适配器”（如图 26-1）。

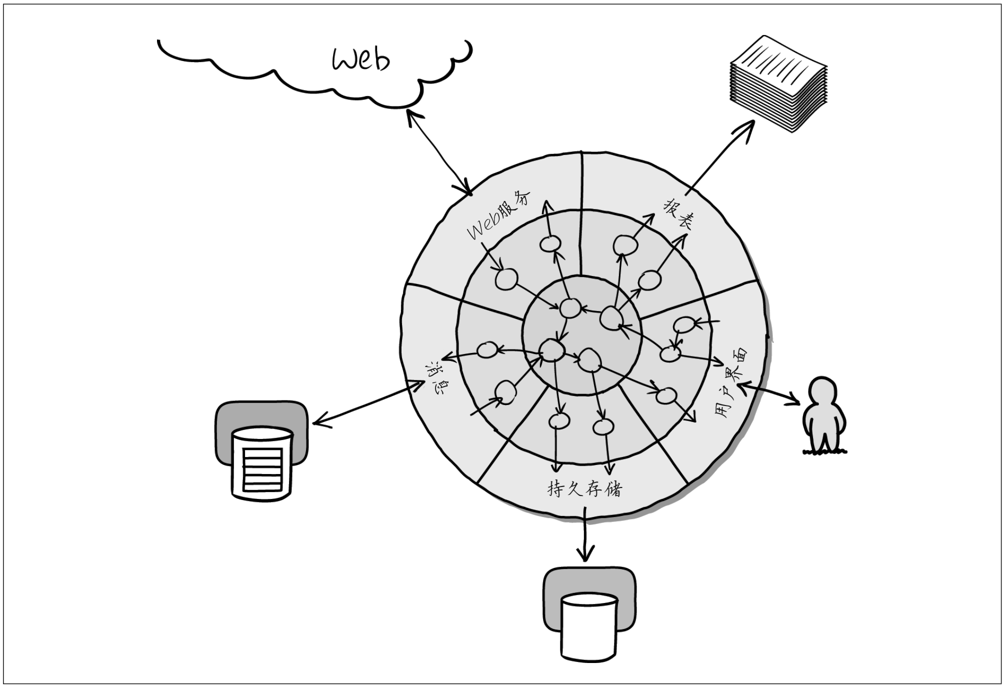


图 26-1：端口和适配器（Nat Pryce 绘制）

其实，第 23 章已经朝端口和适配器架构方案努力了，当时我们发现编写隔离的单元测试要把 ORM 代码从主应用中移除，定义为模型层的辅助函数。

这种模式有时也叫“简洁架构”或“六边形架构”。详情参见本章末尾的扩展阅读部分。

26.4.2 函数式核心，命令式外壳

Gary Bernhardt 更进一步，推荐使用他称为"函数式核心，命令式外壳"的架构。应用的"外壳"是边界交互的地方，遵守命令式编程范式，可以使用整合测试、验收测试检查，如果精简到一定程度，甚至完全不用测试。而应用的核心使用函数式编程范式编写（完全没有副作用），因此可以使用完全隔离、纯粹的单元测试，**根本无须使用驭件**。

这个方案的详细说明，参见 Gary 的演讲，主题为 Boundaries。

26.5 小结

我尝试概述了 TDD 流程涉及的深层次注意事项。经过长年的实践经验积累才能领悟这些观点，所以我非常没资格讨论这些事情。我衷心鼓励你怀疑我所说的一切，尝试不同的方案，听听其他人是怎么说的，找到适合自己的方法。

下面列出了一些扩展阅读资料。

扩展阅读

- "Fast Test, Slow Test"和"Boundaries"
  Gary Bernhardt 分别于 2012 年（"Fast Test, Slow Test"）和 2013 年（"Boundaries"）在 Pycon 中所做的演讲。他制作的视频（Destroy All Software）也值得一看。
- 端口和适配器
  Steve Freeman 和 Nat Pryce 在他们合著的书中提出这种架构。Steve Freeman 一场名为"Test-Driven Development"的演讲对此也做了很好的讨论。还可以阅读 Uncle Bob 对简洁架构的说明（"The Clean Architecture"），以及 Alistair Cockburn 提出六边形架构的文章（"Hexagonal Architecture"）。
- 炽热的岩浆
  Casey Kinsey 提醒，尽量避免和数据库交互（演讲"Writing Fast and Efficient Unit Tests for Django"）。
- 翻转金字塔
  如果项目中运行速度慢的高层测试和单元测试的比值太大，可以使用这种形象的比喻努力翻转比值。
- 整合测试是个骗局
  J.B. Rainsberger 写过一篇著名的文章（"Integrated Tests Are A Scam"），痛斥整合测试，声称它会毁了你的生活。还可以阅读几篇后续文章，尤其是这篇防范验收测试（我叫它功能测试）的文章（"Using Integration Tests Mindfully: A Case Study"），以及这篇分析速度慢的测试是如何扼杀效率的文章（"Part 2: Some Hidden Costs of Integration Tests"）。

- Test-Double 测试维基
  Justin Searls 的在线资源（https://github.com/testdouble/contributing-tests/wiki/Test-Driven-Development）对相关概念做出了准确的定义，还讨论了测试的优缺点，而且总结了各项操作的正确做法。
- 务实的角度
  Martin Fowler（《重构》的作者）提出一种合理平衡的务实方案（http://martinfowler.com/bliki/UnitTest.html）。

> **在不同的测试类型之间正确权衡**
>
> - 务实为本
>   花费大量时间纠结编写何种测试往往得不偿失。最好跟着感觉走，先编写下意识觉得应该编写的测试，然后再根据需要修改。在实践中学习。
> - 关注想从测试中得到什么
>   我们的目标是**正确性、好的设计和快速的反馈循环**。不同类型的测试以不同的方式达到这些目标。表 26-1 列出了一些自问的好问题。
> - 架构很重要
>   架构在某种程度上决定了所需的测试类型。业务逻辑与外部依赖隔离得越好，代码的模块化程度越高，在单元测试、集成测试和端到端测试之间便能达到越好的平衡。

遵从测试山羊的教诲

回过头再看测试山羊。

你可能会说："唉，哈利，大概 17 章之前，测试山羊就没那么有趣了。"请容许我唠叨几句，我要用测试山羊表达一些重要的观点。

测试很难

当我看到"遵从测试山羊的教诲"这句话时，第一印象是它道出了一个事实：测试很难——不是说测试本身难，而是难在**坚持**，一直做下去。

走捷径少写几个测试感觉更容易。而且心理上更难接受测试，因为付出的努力和得到的回报太不成正比。现在花时间编写的测试不会立即显出功效，要等到很久以后才有作用——或许几个月之后避免在重构过程中引入问题，或者升级依赖时捕获回归。或许测试会以一种很难衡量的方式回报你，促使你写出设计更好的代码，但你却认为就算没有测试也能写出如此优雅的代码。

为本书编写测试框架时，（http://github.com/hjwp/Book-TDD-Dev-Python/tree/master/tests）我自己也开始犯这种错误了。书中的代码很复杂，所以本身也有测试，但我偷懒了，测试覆盖度并不理想，现在我后悔了，因为测试写得笨拙又丑陋（好了，我已经开源这个测试框架，指责、嘲笑我吧）。

让CI构建始终能通过

需要真正付出心力的另一个领域是持续集成。读过第 24 章我们知道，CI 构建有时会出现意料之外的奇怪问题。出现这种问题时，如果觉得在自己的设备中正常就行，很容易放任

不管。但是，如果不小心，你会开始容忍 CI 中有失败的测试组件，久而久之，CI 构建便失去了意义。若想再次让 CI 构建运行起来，工作量更大。千万别落入这个圈套。只要坚持，终究会找到测试失败的原因，而且能找到解决的方法，再次让构建通过，发挥决断作用。

像重视代码一样重视测试

别再把测试看成真正的代码的陪衬，把它当作你在开发的产品的一部分，精心雕琢，注重美观，就算发布出去也不会羞于面对众人检视。这么想有助于你接受测试。

做测试的原因有很多：可能是测试山羊告诉你要做；可能是你知道就算不能立即得到回报，但终究会得到；可能是出于责任感、职业素养、强迫症或者想挑战自己；还可能是因为测试值得实践。但终极原因是，测试让软件开发变得更有乐趣。

别忘了给吧台服务员小费

没有 O'Reilly Media 出版社的支持，我不可能写成这本书。如果你看的是在线免费版，希望你能考虑买一本。如果你自己不需要，或许可以作为礼物送给朋友。

别见外

希望你喜欢这本书。请一定要和我联系，告诉我你的想法！

- @hjwp（Ttwitter）
- obeythetestinggoat@gmail.com

附录A

PythonAnywhere

本书假设你在自己的电脑上运行 Python 和编程。当然，这不是如今做 Python 编程的唯一方式，你也可以使用在线平台，例如 PythonAnywhere（碰巧，我就在这工作）。

在阅读本书的过程中，你可以使用 PythonAnywhere，但是要做些调整和修改：测试时要设置一个 Web 应用而不是用测试服务器；要使用 Xvfb 运行功能测试；而且读到部署那几章时，要升级到付费账户。虽然可以这么做，但还是使用自己的电脑更方便。

倘若你确实想试一下，可以参照下文所述去做。

如果你还没有 PythonAnywhere 的账户，先要注册一个，免费的就行。

然后，在控制台页面启动一个 Bash Console。大多数工作都将在这个控制台中完成。

A.1　使用Xvfb在Firefox中运行Selenium会话

首先要知道，PythonAnywhere 是只有终端的环境，所以没有显示器就无法打开 Firefox。但我们可以使用虚拟显示器。

读第 1 章编写第一个测试时，你会发现无法正常运行。第一个测试如下所示，可以在 PythonAnywhere 提供的编辑器中输入：

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://localhost:8000')
assert 'Django' in browser.title
```

但（在 Bash 终端）运行时会看到如下错误：

```
(superlists)$ python functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: 'geckodriver' executable
needs to be in PATH.
```

这是因为 PythonAnywhere 所用的 Firefox 是旧版本，不需要 Geckodriver。但是，我们要把 Selenium 3 换成 Selenium 2：

```
(superlists) $ pip install "selenium<3"
Collecting selenium<3
Installing collected packages: selenium
  Found existing installation: selenium 3.4.3
    Uninstalling selenium-3.4.3:
      Successfully uninstalled selenium-3.4.3
Successfully installed selenium-2.53.6
```

现在又会遇到一个问题：

```
(superlists)$ python functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: The browser appears to
have exited before we could connect. If you specified a log_file in the
FirefoxBinary constructor, check it for details.
```

Firefox 无法启动，因为没有运行所需的显示器，毕竟 PythonAnywhere 是服务器环境。解决方法是使用 Xvfb（X Virtual Framebuffer 的简称）。在没有真正的显示器的服务器中，Xvfb 会启动一个“虚拟”显示器，供 Firefox 使用。

`xvfb-run` 命令的作用是，在 Xvfb 中执行下一个命令。使用这个命令就会看到预期失败：

```
(superlists)$ xvfb-run -a python functional_tests.py
Traceback (most recent call last):
File "tests.py", line 11, in <module>
assert 'Django' in browser.title
AssertionError
```

记住，只要想运行功能测试，就要使用 `xvfb-run -a` 命令。

A.2　以PythonAnywhere Web应用的方式安装Django

随后要使用 `django-admin.py startproject` 命令创建 Django 项目。但是，不要使用 `manage.py runserver` 启动本地开发服务器，我们将把网站设置为真正的 PythonAnywhere Web 应用。

打开“Web”标签页，点击按钮添加一个新 Web 应用。选择“Manual configuration”（手动配置），然后再选“Python 3.4”。

在下一个界面中输入虚拟环境的名称（“superlists”），提交后，会自动补全为 /home/yourusername/.virtualenvs/superlists。

最后，找到**编辑 wsgi 文件**的链接，找出针对 Django 的那一部分，去掉注释。点击“Save”，再点击“Reload”，刷新 Web 应用。

从现在开始，不要在控制台中运行 localhost:8000 上的测试服务器，你可以使用 PythonAnywhere 为 Web 应用分配的真实 URL 了：

```
browser.get('http://my-username.pythonanywhere.com')
```

> 每次修改代码后都要点击“Reload Web App”（重新加载 Web 应用）按钮，更新网站。

效果更好！[1] 在第 7 章换用 `LiveServerTestCase` 和 `self.live_server_url` 之前，在 PythonAnywhere 中必须这样指向功能测试，而且每次运行功能测试之前都要点击“Reload”。

A.3　清理/tmp目录

Selenium 和 Xvfb 会在 /tmp 目录中留下很多垃圾，如果关闭的方式不优雅，情况会更糟（所以前文才要使用 `try/finally` 语句）。

遗留的东西太多，可能会用完存储配额。所以要经常清理 /tmp 目录：

```
$ rm -rf /tmp/*
```

A.4　截图

在第 5 章中，我建议使用 `time.sleep` 在功能测试运行的过程中暂停一会儿，这样才能在屏幕上看到 Selenium 浏览器。在 PythonAnywhere 做不到这一点，因为浏览器运行在虚拟显示器中。不过你可以检查线上网站，或者别管应该看到什么，相信我说的就行了。

对运行在虚拟显示器中的测试做视觉检查，最好的方法是使用截图。如果你想知道怎么做，看一下第 24 章，那里有一些示例代码。

A.5　关于部署

读到第 9 章时，你可以选择继续使用 PythonAnywhere，也可以选择学习如何配置真实的服务器。我建议选择后者，因为这样能学到更多。

注 1：也**可以**在终端里运行开发服务器，但有个问题，PythonAnywhere 的终端不一定运行在同一台服务器中，所以无法保证运行测试的终端和运行服务器的终端是同一个。而且，如果在终端里运行服务器，没有简单的方法视觉检查网站的外观。

如果想一直使用 PythonAnywhere，可以再注册一个 PythonAnywhere 账户，用作过渡网站（作弊嫌疑很大）。或者，为现有账户再添加一个域名。但部署那一章的内容就用不到了（在 PythonAnywhere 上无须 Nginx、Gunicorn 或域套接字）。

遇到下列情形之一时，你需要一个付费账户。

- 如果过渡网站不使用 PythonAnywhere 提供的域名。
- 如果不想在 PythonAnywhere 提供的域名上运行功能测试（因为别的域名不在白名单上）。
- 读到第 11 章，如果想使用 PythonAnywhere 账户运行 Fabric（因为需要 SSH）。

如果你想“作弊”，可以在现有的 Web 应用中以“过渡”模式运行功能测试，并跳过涉及 Fabric 的部分——这算是一种妥协吧。其实，你可以先升级账户，然后立即取消，在 30 天保障期内申请退款。

> 如果阅读本书时你从头至尾都使用 PythonAnywhere，我很想听听你是怎么做到的。请给我发电子邮件，地址是 obeythetestinggoat@gmail.com。

附录B

基于类的Django视图

本附录接续第 15 章。第 15 章实现了 Django 表单的验证功能，还重构了视图。结束时，视图仍然使用函数实现。

不过，Django 领域现在流行使用基于类的视图（Class-Based View，CBV）。在这个附录中，我们要重构应用，把视图函数改写成基于类的视图。更确切地说，我们要尝试使用基于类的**通用视图**（Class-Based Generic View，CBGV）。

B.1 基于类的通用视图

基于类的视图和基于类的**通用**视图有个区别。基于类的视图（class-based view，CBV）只是定义视图函数的另一种方式，对视图要做的事情没有太多假设，和视图函数相比主要的优势是可以创建子类。不过也要付出一定代价，基于类的视图比传统的基于函数的视图可读性差（这一点有争论）。**普通的** CBV 的作用是让多个视图重用相同的逻辑，因为我们想遵守 DRY 原则。如果使用基于函数的视图，重用逻辑要使用辅助函数或修饰器。理论上，使用类实现更优雅。

基于类的**通用**视图也是一种基于类的视图，但它尝试为常见操作提供现成的解决方案，例如从数据库中获取对象后传入模板，获取一组对象，使用 `ModelForm` 保存 POST 请求中用户输入的数据，等等。看起来现在需要的就是这种视图，不过稍后就会发现魔鬼藏在细节中。

这里我想说，我并不常用任何一种基于类的视图。我完全能看到这种视图的合理之处，而且在 Django 应用中有很多地方都非常适合使用 CBGV。但是，只要需求稍微高一点儿，例如想使用多个模型，就会发现基于类的视图比传统的视图函数难读得多（这一点也有争论）。

不过，因为必须使用基于类的视图提供的几个定制选项，通过这种实现方式能学到很多这种视图的工作方式，以及如何为这种视图编写单元测试。

我希望为基于函数的视图编写的单元测试也能正常测试基于类的视图。看一下具体该怎么做。

B.2 使用FormView实现首页

网站的首页只是在模板中显示一个表单：

lists/views.py

```
def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

看过可选视图（https://docs.djangoproject.com/en/1.11/ref/class-based-views/）之后，我们发现 Django 提供了一个通用视图，叫 `FormView`。看一下怎么用：

lists/views.py (ch31l001)

```
from django.views.generic import FormView
[...]

class HomePageView(FormView):
    template_name = 'home.html'
    form_class = ItemForm
```

指定想使用哪个模板和表单。然后，只需更新 urls.py，把含有 `lists.views.home_page` 那行代码改成：

superlists/urls.py (ch31l002)

```
[...]
urlpatterns = [
    url(r'^$', list_views.HomePageView.as_view(), name='home'),
    url(r'^lists/', include(list_urls)),
]
```

运行所有测试确认，这很简单：

```
$ python manage.py test lists
[...]

Ran 34 tests in 0.119s
OK

$ python manage.py test functional_tests
[...]
Ran 5 tests in 15.160s
OK
```

目前为止一切顺利。把一行代码的视图函数换成有两行代码的类，而且可读性依然不错。现在是提交的好时机。

B.3　使用form_valid定制CreateView

下面改写新建清单的视图，也就是 `new_list` 函数。现在这个视图如下所示：

lists/views.py

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

浏览可用的 CBGV 列表之后，发现需要的或许是 `CreateView`，而且知道要使用 `ItemForm` 类，下面看一下具体该怎么做，以及测试能否提供帮助：

lists/views.py (ch31l003)

```
from django.views.generic import FormView, CreateView
[...]

class NewListView(CreateView):
    form_class = ItemForm

def new_list(request):
    [...]
```

我要在 views.py 中保留原来的视图函数，这样才能从中复制代码，等一切都能正常运行之后再删除。这么做没什么危害，只要修改 URL 映射就行。这一次要这么改：

lists/urls.py (ch31l004)

```
[...]
urlpatterns = [
    url(r'^new$', views.NewListView.as_view(), name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
]
```

然后运行测试。有 6 个错误：

```
$ python manage.py test lists
[...]

ERROR: test_can_save_a_POST_request (lists.tests.test_views.NewListTest)
TypeError: save() missing 1 required positional argument: 'for_list'

ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'
```

```
ERROR: test_for_invalid_input_renders_home_template
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'

ERROR: test_invalid_list_items_arent_saved (lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'

ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
TypeError: save() missing 1 required positional argument: 'for_list'

ERROR: test_validation_errors_are_shown_on_home_page
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'


FAILED (errors=6)
```

先解决前 3 个——设定模板应该就可以了吧?

lists/views.py (ch31l005)

```
class NewListView(CreateView):
    form_class = ItemForm
    template_name = 'home.html'
```

现在只剩两个错误了，可以看出，这两个错误都发生在通用视图的 `form_valid` 方法中。这个方法可以重新定义来定制 CBGV 的行为。从这个方法的名字可以看出，视图认为表单中的数据有效之后才会运行这个方法。可以从以前的视图函数中把 `if form.is_valid():` 之后的代码复制过来：

lists/views.py (ch31l006)

```
class NewListView(CreateView):
    template_name = 'home.html'
    form_class = ItemForm

    def form_valid(self, form):
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
```

这样测试就能全部通过了：

```
$ python manage.py test lists
Ran 34 tests in 0.119s
OK
$ python manage.py test functional_tests
Ran 5 tests in 15.157s
OK
```

而且，为了遵守 DRY 原则，**可以**使用 CBV 的主要优势之一 ——继承，节省两行代码：

lists/views.py (ch31l007)

```
class NewListView(CreateView, HomePageView):

    def form_valid(self, form):
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
```

测试应该仍能全部通过：

```
OK
```

> 其实在面向对象编程中这么做并不好。继承意味着“是一个什么”这种关系，但是说新建清单视图“是一个”首页视图或许没有什么意义。所以，或许最好别这么做。

不管做没做最后一步，你觉得和以前的版本相比怎么样？我觉得还不错。不用写样板代码了，而且视图代码还相当易读。目前，CBGV 得了一分，和基于函数的视图平局。

B.4　一个更复杂的视图，既能查看清单，也能向清单中添加待办事项

我做了**好多**尝试才写出这个视图。不得不说，虽然测试能告诉我做得对不对，但是在寻找实现步骤的过程中并不能给出实质帮助，大多数情况下我都在反复实验，尝试使用 `get_context_data` 和 `get_form_kwargs` 等函数。

不过，在实现的过程中我意识到一件事：编写多个只测试一件事的测试很重要。所以又回头重写了第 10~12 章的部分内容。

B.4.1　测试有指引作用，但时间不长

下面是一种实现方式。首先，我觉得需要使用 `DetailView`，显示对象的详情：

lists/views.py (ch31l009)

```
from django.views.generic import FormView, CreateView, DetailView
[...]

class ViewAndAddToList(DetailView):
    model = List
```

然后在 urls.py 中设置：

lists/urls.py (ch31l010)

```
    url(r'^(\d+)/$', views.ViewAndAddToList.as_view(), name='view_list'),
```

测试的结果为：

```
[...]
AttributeError: Generic detail view ViewAndAddToList must be called with either
an object pk or a slug.


FAILED (failures=5, errors=6)
```

失败消息的意思并不明确，在谷歌中搜索一番之后我才知道要使用正则表达式具名捕获组：

lists/urls.py (ch31l011)

```
@@ -3,6 +3,6 @@ from lists import views

 urlpatterns = [
     url(r'^new$', views.NewListView.as_view(), name='new_list'),
-    url(r'^(\d+)/$', views.view_list, name='view_list'),
+    url(r'^(?P<pk>\d+)/$', views.ViewAndAddToList.as_view(), name='view_list')
 ]
```

接下来出现的错误中有一个相当有帮助：

```
[...]
django.template.exceptions.TemplateDoesNotExist: lists/list_detail.html

FAILED (failures=5, errors=6)
```

这个很容易解决：

lists/views.py (ch31l012)

```
class ViewAndAddToList(DetailView):
    model = List
    template_name = 'list.html'
```

这次改动后，只剩 5 个失败和 2 个错误了：

```
[...]
ERROR: test_displays_item_form (lists.tests.test_views.ListViewTest)
KeyError: 'form'

FAILED (failures=5, errors=2)
```

B.4.2　现在不得不反复实验

我发现这个视图不仅要显示对象的详情，还要能新建对象。所以这个视图要继承 `DetailView` 和 `CreateView`，可能还要添加 `form-class` 属性：

lists/views.py (ch31l013)

```
class ViewAndAddToList(DetailView, CreateView):
    model = List
    template_name = 'list.html'
    form_class = ExistingListItemForm
```

但是这么改，得到了很多错误：

```
[...]
TypeError: __init__() missing 1 required positional argument: 'for_list'
```

而且那个 `KeyError: 'form'` 错误还在。

现在，错误消息没什么用了，而且下一步该做什么也不明确。我不得不反复实验。不过，测试仍能告诉我做得对还是把情况变得更糟了。

首先尝试使用 `get_form_kwargs`，但没什么用，不过我发现可以使用 `get_form`：

lists/views.py (ch31l014)

```
    def get_form(self):
        self.object = self.get_object()
        return self.form_class(for_list=self.object, data=self.request.POST)
```

但必须给 `self.object` 赋值才能使用 `get_form`，对此我一直心里不安。不过现在只剩 3 个错误了，显然是因为表单还没传入模板。

```
django.core.exceptions.ImproperlyConfigured: No URL to redirect to.  Either
provide a url or define a get_absolute_url method on the Model.
```

B.4.3　测试再次发挥作用

对最后的这个失败，测试又变有用了。解决的方法很简单，在 `Item` 类中定义 `get_absolute_url` 方法，让待办事项指向所属的清单页面即可：

lists/models.py (ch31l015)

```
class Item(models.Model):
    [...]

    def get_absolute_url(self):
        return reverse('view_list', args=[self.list.id])
```

B.4.4　这是最终结果吗

最终写出的视图类如下所示：

lists/views.py

```
class ViewAndAddToList(DetailView, CreateView):
    model = List
    template_name = 'list.html'
    form_class = ExistingListItemForm

    def get_form(self):
        self.object = self.get_object()
        return self.form_class(for_list=self.object, data=self.request.POST)
```

B.5　新旧版对比

比较一下旧版和新版：

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

虽然新版代码从 9 行缩减到 7 行，但我还是觉得基于函数的视图稍微容易理解一点儿，因为旧版没隐藏那么多细节，毕竟“明确表述比含糊其辞强”，这是 Python 的禅理。我的意思是，谁知道 `SingleObjectMixin` 是什么呢？而且更讨厌的是，如果在 `get_form` 方法中不给 `self.object` 赋值，整个类都不能用。这一点太烦人。

不过，我猜有些人喜欢这种实现方式。

B.6　为CBGV编写单元测试有最佳实践吗

实现这个视图类之后，我发现有些单元测试有点儿太关注高层。这并不意外，因为使用 Django 测试客户端的视图测试或许更应该叫整合测试。

测试会告诉我做得对不对，但不能始终提示具体应该怎么修正错误。

有时，我想知道有没有一种编写测试的方法，更贴近实现方式，例如这么编写测试：

lists/tests/test_views.py

```
def test_cbv_gets_correct_object(self):
    our_list = List.objects.create()
    view = ViewAndAddToList()
    view.kwargs = dict(pk=our_list.id)
    self.assertEqual(view.get_object(), our_list)
```

但这么做有个问题，必须对 Django CBV 的内部机理有一定了解，才能正确设定这种测试。而且最后还是会被复杂的继承体系弄得十分糊涂。

B.7　记住：编写多个只有一个断言的隔离视图测试有所帮助

在这个附录中我得出一个结论：编写多个简短的单元测试比编写少量含有很多断言的测试有用得多。

看看这个庞大的测试：

lists/tests/test_views.py

```
    def test_validation_errors_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(List.objects.all().count(), 0)
        self.assertEqual(Item.objects.all().count(), 0)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

它肯定没有下面这 3 个单独的测试有用：

lists/tests/test_views.py

```
        def test_invalid_input_means_nothing_saved_to_db(self):
            self.post_invalid_input()
            self.assertEqual(List.objects.all().count(), 0)
            self.assertEqual(Item.objects.all().count(), 0)

        def test_invalid_input_renders_list_template(self):
            response = self.post_invalid_input()
            self.assertTemplateUsed(response, 'list.html')

        def test_invalid_input_renders_form_with_errors(self):
            response = self.post_invalid_input()
            self.assertIsinstance(response.context['form'], ExistingListItemForm)
            self.assertContains(response, escape(empty_list_error))
```

因为对前一种方式来说，如果靠前的断言失败了，后面的断言就不会执行。所以，如果视图不小心把 POST 请求中的无效数据存入数据库，前面的断言会失败，这样就无法确认使用的模板是否正确以及有没有渲染表单。使用后一种方式则能更轻易地分辨出到底哪一部分能用，哪一部分不能用。

从 CBGV 中学到的经验

- 基于类的通用视图可以做任何事
  虽然不一定总是知道到底怎么回事，但使用基于类的视图几乎可以做任何事。
- 只有一个断言的单元测试有助于重构
  有单元测试检查什么可用什么不可用，使用不同的基本范式修改视图的实现方式就容易多了。

附录C

使用Ansible配置服务器

用 Fabric 自动把新版源码部署到服务器上，但配置新服务器的过程以及更新 Nginx 和 Gunicorn 配置文件的操作都还是手动完成。

这类操作越来越多地交给“配置管理”或“持续部署”工具完成。其中，Chef 和 Puppet 最受欢迎，而在 Python 领域则是 Salt 和 Ansible。

在这些工具中，Ansible 最容易上手，只需两个文件就可以使用：

```
pip2 install --user ansible # 可惜只能用Python 2
```

清单文件 deploy_tools/inventory.ansible 定义可以在哪些服务器中运行：

deploy_tools/inventory.ansible

```
[live]
superlists.ottg.eu ansible_become=yes ansible_ssh_user=elspeth

[staging]
superlists-staging.ottg.eu ansible_become=yes ansible_ssh_user=elspeth

[local]
localhost ansible_ssh_user=root ansible_ssh_port=6666 ansible_host=127.0.0.1
```

（local 条目只是个示例，在我的设备中是个 Virtualbox 虚拟主机，并且为 22 和 80 端口设定了端口转发。）

C.1　安装系统包和Nginx

另一个文件是“脚本”（playbook），定义在服务器中做什么。这个文件的内容使用 YAML 句法编写：

deploy_tools/provision.ansible.yaml

```
---

- hosts: all

  vars:
      host: "{{ inventory_hostname }}"

  tasks:

    - name: Deadsnakes PPA to get Python 3.6
      apt_repository:
        repo='ppa:fkrull/deadsnakes'
    - name: make sure required packages are installed
      apt: pkg=nginx,git,python3.6,python3.6-venv state=present

    - name: allow long hostnames in nginx
      lineinfile:
        dest=/etc/nginx/nginx.conf
        regexp='(\s+)#? ?server_names_hash_bucket_size'
        backrefs=yes
        line='\1server_names_hash_bucket_size 64;'

    - name: add nginx config to sites-available
      template: src=./nginx.conf.j2 dest=/etc/nginx/sites-available/{{ host }}
      notify:
          - restart nginx

    - name: add symlink in nginx sites-enabled
      file:
          src=/etc/nginx/sites-available/{{ host }}
          dest=/etc/nginx/sites-enabled/{{ host }}
          state=link
      notify:
          - restart nginx
```

`inventory_hostname` 变量是目标服务器的域名。为了方便引用，我在 `vars` 部分把它重命名成了“host”。

在“tasks”部分，使用 `apt` 安装所需的软件，再使用正则表达式替换 Nginx 配置，允许使用长域名，然后使用模板创建 Nginx 配置文件。这个模板由第 9 章保存在 deploy_tools/nginx.template.conf 中的模板文件修改而来，不过现在指定使用一种模板引擎——Jinja2，和 Django 的模板句法很像：

deploy_tools/nginx.conf.j2

```
server {
    listen 80;
    server_name {{ host }};

    location /static {
        alias /home/{{ ansible_ssh_user }}/sites/{{ host }}/static;
    }
```

```
    location / {
        proxy_set_header Host {{ host }};
        proxy_pass http://unix:/tmp/{{ host }}.socket;
    }
}
```

C.2　配置Gunicorn，使用处理程序重启服务

脚本剩余的内容如下：

deploy_tools/provision.ansible.yaml

```
    - name: write gunicorn service script
      template:
          src=./gunicorn.service.j2
          dest=/etc/systemd/system/gunicorn-{{ host }}.service
      notify:
          - restart gunicorn

  handlers:
    - name: restart nginx
      service: name=nginx state=restarted

    - name: restart gunicorn
      systemd:
          name=gunicorn-{{ host }}
          daemon_reload=yes
          enabled=yes
          state=restarted
```

创建 Gunicorn 配置文件还要使用模板：

deploy_tools/gunicorn.service.j2

```
[Unit]
Description=Gunicorn server for {{ host }}

[Service]
User={{ ansible_ssh_user }}
WorkingDirectory=/home/{{ ansible_ssh_user }}/sites/{{ host }}/source
Restart=on-failure
ExecStart=/home/{{ ansible_ssh_user }}/sites/{{ host }}/virtualenv/bin/gunicorn \
    --bind unix:/tmp/{{ host }}.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application

[Install]
WantedBy=multi-user.target
```

然后定义两个处理程序，重启 Nginx 和 Gunicorn。Ansible 很智能，如果多个步骤都调用同

一个处理程序，它会等前一个执行完再调用下一个。

这样就行了！执行配置操作的命令如下：

```
ansible-playbook -i inventory.ansible provision.ansible.yaml --limit=staging --ask-become-pass
```

详细信息参阅 Ansible 的文档。

C.3 接下来做什么

我只是简单介绍了 Ansible 的功能。部署过程的自动化程度越高，你对部署也越自信。接下来，你可以完成下面几件事。

C.3.1 把Fabric执行的部署操作交给Ansible

已经看到 Ansible 可以帮助完成配置过程中的某些操作，其实它可以完成几乎所有部署操作。你可以试一下，看能否扩写脚本把当前 Fabric 部署脚本中的所有操作都交给 Ansible 完成，包括必要情况下重启时发出提醒。

C.3.2 使用Vagrant搭建本地虚拟主机

在过渡网站中运行测试能让我们相信网站上线后也能正常运行。不过也可以在本地设备中使用虚拟主机完成这项操作。

下载 Vagrant 和 Virtualbox，看你能否使用 Vagrant 在自己的电脑中搭建一个开发服务器，以及使用 Ansible 脚本把代码部署到这个服务器中。设置功能测试运行程序，让功能测试能在本地虚拟主机中运行。

如果在团队中工作，编写一个 Vagrant 配置脚本特别有用，因为它能帮助新加入的开发者搭建和你们使用的一模一样的服务器。

附录 D

测试数据库迁移

Django-migrations 及其前身 South 已经出现好多年了，所以一般没必要测试数据库迁移。但有时我们会使用一种危险的迁移，即引入新的数据完整性约束。我第一次在过渡环境中运行这种迁移脚本时，遇到了一个错误。

在大型项目中，如果有敏感数据，在生产数据中执行迁移之前，你可能想先在一个安全的环境中测试，增加一些自信。你可以在本书开发的示例应用中先练习一下。

测试迁移的另一个常见原因是测速——执行迁移时经常要把网站下线，而且如果数据集较大，用时并不短。所以最好提前知道迁移要执行多长时间。

D.1　尝试部署到过渡服务器

在第 17 章，当我第一次尝试部署新添加的验证约束条件时，遇到了如下问题：

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
[...]
Running migrations:
  Applying lists.0005_list_item_unique_together...Traceback (most recent call
last):
  File "/usr/local/lib/python3.6/dist-packages/django/db/backends/utils.py",
line 61, in execute
    return self.cursor.execute(sql, params)
  File
"/usr/local/lib/python3.6/dist-packages/django/db/backends/sqlite3/base.py",
line 475, in execute
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: columns list_id, text are not unique
[...]
```

情况是这样，数据库中某些现有的数据违反了完整性约束条件，所以当我尝试应用约束条件时，数据库表达了不满。

为了处理这种问题，需要执行“数据迁移”。首先，要在本地搭建一个测试环境。

D.2　在本地执行一个用于测试的迁移

使用线上数据库的副本测试迁移。



测试时使用真实数据一定要小心小心再小心。例如，数据中可能有客户的真实电子邮件地址，但你并不想不小心给他们发送一堆测试邮件。我可是栽过跟头的。

D.2.1　输入有问题的数据

在线上网站中新建一个清单，输入一些重复的待办事项，如图 D-1 所示。

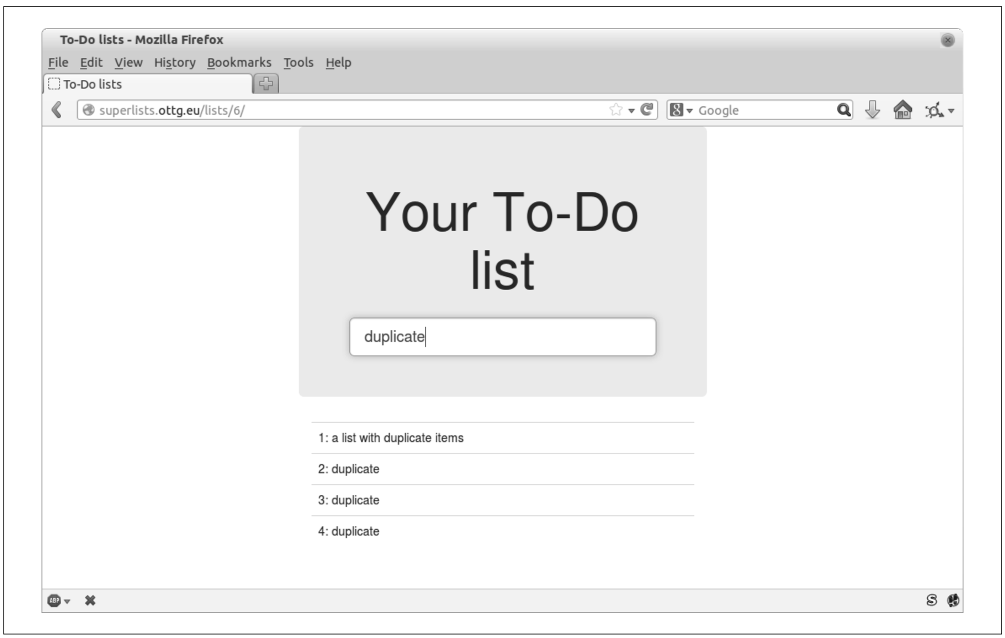


图 D-1：一个清单，待办事项有重复

D.2.2　从线上网站中复制测试数据

从线上网站中复制数据库：

```
$ scp elspeth@superlists.ottg.eu:\
/home/elspeth/sites/superlists.ottg.eu/database/db.sqlite3 .
```

```
$ mv ../database/db.sqlite3 ../database/db.sqlite3.bak
$ mv db.sqlite3 ../database/db.sqlite3
```

D.2.3　确认的确有问题

现在，本地有一个还未执行迁移的数据库，而且数据库中有一些问题数据。如果尝试执行 `migrate` 命令，会看到一个错误：

```
$ python manage.py migrate --migrate
python manage.py migrate
Operations to perform:
[...]
Running migrations:
[...]
  Applying lists.0005_list_item_unique_together...Traceback (most recent call
last):
[...]
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: columns list_id, text are not unique
```

D.3　插入一个数据迁移

数据迁移是一种特殊的迁移，目的是修改数据库中的数据，而不是变更模式。应用完整性约束之前，先要执行一次数据迁移，把重复数据删除。具体方法如下：

```
$ git rm lists/migrations/0005_list_item_unique_together.py
$ python manage.py makemigrations lists --empty
Migrations for 'lists':
  0005_auto_20140414_2325.py:
$ mv lists/migrations/0005_*.py lists/migrations/0005_remove_duplicates.py
```

有关数据迁移的详情，请参阅 Django 文档。下面是修改现有数据的方法：

lists/migrations/0005_remove_duplicates.py

```
# encoding: utf8
from django.db import models, migrations

def find_dupes(apps, schema_editor):
    List = apps.get_model("lists", "List")
    for list_ in List.objects.all():
        items = list_.item_set.all()
        texts = set()
        for ix, item in enumerate(items):
            if item.text in texts:
                item.text = '{} ({})'.format(item.text, ix)
                item.save()
            texts.add(item.text)


class Migration(migrations.Migration):
```

```
    dependencies = [
        ('lists', '0004_item_list'),
    ]

    operations = [
        migrations.RunPython(find_dupes),
    ]
```

重新创建以前的迁移

使用 makemigrations 重新创建以前的迁移，确保这是第 6 个迁移，而且还明确依赖于 0005，即那个数据迁移：

```
$ python manage.py makemigrations
Migrations for 'lists':
  0006_auto_20140415_0018.py:
    - Alter unique_together for item (1 constraints)
$ mv lists/migrations/0006_* lists/migrations/0006_unique_together.py
```

D.4　一起测试这两个迁移

现在可以在线上数据中测试了：

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
[...]
```

还要重启服务器中的 Gunicorn 服务：

```
elspeth@server:$ sudo systemctl restart gunicorn-superlists.ottg.eu
```

然后可以在过渡网站中运行功能测试：

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
[...]
....
 ---------------------------------------------------------------------
Ran 4 tests in 17.308s

OK
```

看起来一切正常。现在部署到线上服务器：

```
$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[...]
```

最后，执行 git add lists/migrations、git commit 等命令。

D.5 小结

这个练习的主要目的是编写一个数据迁移，在一些真实的数据中测试。当然，这只是测试迁移的千万种方式之一。你还可以编写自动化测试，比较运行迁移前后数据库中的内容，确认数据还在。也可以为数据迁移中的辅助函数单独编写单元测试。你可以再多花点儿时间统计迁移所用的时间，然后实验多种提速的方法，例如，把迁移的步骤分得更细或更笼统。

记住，这种需求很少见。根据我的经验，我使用的迁移有 99% 都不需要测试。不过，在你的项目中可能需要。希望读过这个附录之后，你知道怎么着手测试迁移。

关于测试数据库迁移

- 小心引入约束的迁移
  99% 的迁移都没问题，但是如果迁移为现有的列引入了新的约束条件，就像前面那个例子，一定要小心。
- 测试迁移的执行速度
  一旦项目变大，你就应该考虑测试迁移所用的时间。执行数据库迁移时往往要下线网站，因为修改模式可能要锁定数据表（取决于使用的数据库种类），直到操作完成为止。所以最好在过渡网站中测试迁移要用多长时间。
- 使用生产数据的副本时要格外小心
  为了测试迁移，要在过渡网站的数据库中填充与生产数据等量的数据。具体怎么做不在本书范畴之内，不过我要提醒一下：如果直接转储生产数据库导入过渡网站的数据库中，一定要**十分**小心，因为生产数据中包含真实客户的详细信息。有一次在过渡服务器中自动处理刚导入的生产数据副本时，我不小心发送了好几百张错误的发票。那个下午过得可不愉快。

附录 E

行为驱动开发

行为驱动开发（behaviour-driven development，BDD）我用得不多，算不上什么专家，但就目前我所接触的，我认为值得简要介绍一下。本附录将使用 BDD 工具转换一些“常规的”功能测试。

E.1　BDD是什么

严格来说，BDD 是一种方法论，而不是工具集。BDD 测试的是应用的行为，确定是否与我们期望用户看到的一致。因此，本书展示的部分基于 Selenium 的功能测试其实也可以称为 BDD。

但这种测试方法通常与 BDD 采用的特定工具集联系紧密，其中最重要的是 Gherkin 句法，这是编写功能测试（或验收测试）的 DSL，对人类而言可读性高。Gherkin 源自 Ruby 世界，与名为 Cucumber 的测试运行程序捆绑。

在 Python 世界，有几个等效的测试运行工具，例如 Lettuce 和 Behave。目前为止，只有 Behave 兼容 Python 3，因此我们将使用它。此外，还将使用一个插件 behave-django。

> **获取示例代码**
>
> 我将使用第 22 章的示例。待办事项清单网站已经具备基本功能，我们想添加一个新功能：已登录用户应该能在某个地方查看自己编制的清单。在此之前，所有清单都是匿名创建的。
>
> 如果你一直跟着本书操作，我假设你能跳回到那一点。如果想从我的仓库中拉取，要使用 chapter_17 分支。

E.2 基本的准备工作

下面为 BDD 特性描述创建一个 features 目录，在里面再添加一个 steps 目录（马上告诉你它的作用），并为这个新功能创建一个占位文件：

```
$ mkdir -p features/steps
$ touch features/my_lists.feature
$ touch features/steps/my_lists.py
$ tree features
features
├── my_lists.feature
└── steps
    └── my_lists.py
```

然后安装 behave-django，并把它添加到 settings.py 中：

```
$ pip install behave-django
```

superlists/settings.py

```
--- a/superlists/settings.py
+++ b/superlists/settings.py
@@ -40,6 +40,7 @@ INSTALLED_APPS = [
     'lists',
     'accounts',
     'functional_tests',
+    'behave_django',
 ]
```

最后运行 python manage.py，做健全性检查：

```
$ python manage.py behave
Creating test database for alias 'default'...
0 features passed, 0 failed, 0 skipped
0 scenarios passed, 0 failed, 0 skipped
0 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
Destroying test database for alias 'default'...
```

E.3 使用Gherkin句法以“特性描述”的形式编写功能测试

目前，我们的功能测试使用人类可读的注释描述新功能，这叫用户故事，其间穿插着执行故事中每一步的 Selenium 代码。

BDD 要求严格区分这二者：先使用 Gherkin 句法（有时让人觉得拗口）编写人类可读的故事，这叫“特性描述”（feature）；然后，把每一行 Gherkin 代码映射到一个函数上，那个函数包含实现那一“步”（step）所需的 Selenium 代码。

“My Lists”页面这个新功能的特性描述可以写成下面这样：

```
Feature: My Lists
    As a logged-in user
    I want to be able to see all my lists in one page
    So that I can find them all after I've written them

    Scenario: Create two lists and see them on the My Lists page

        Given I am a logged-in user

        When I create a list with first item "Reticulate Splines"
            And I add an item "Immanentize Eschaton"
            And I create a list with first item "Buy milk"

        Then I will see a link to "My lists"

        When I click the link to "My lists"
        Then I will see a link to "Reticulate Splines"
        And I will see a link to "Buy milk"

        When I click the link to "Reticulate Splines"
        Then I will be on the "Reticulate Splines" list page
```

E.3.1 As-a/I want to/So that

顶部是“As-a/I want to/So that”子句。这部分是可选的，没有对应的可执行代码。这只是一种形式，让用户知道“你是谁、你想做什么”，有助于团队成员理解每个功能的背景。

E.3.2 Given/When/Then

“Given/When/Then”是 BDD 测试的真正核心。这三部分与单元测试的“设置 – 使用 – 断言”模式匹配，分别表示设置 / 假设阶段、使用 / 行动阶段以及断言 / 观察阶段。详情参见 Cucumber 维基页面（https://github.com/cucumber/cucumber/wiki/Given-When-Then）。

E.3.3 并不始终完美契合

如你所见，用户故事不是总能完美分成这三步的。我们可以使用 `And` 子句扩充步骤，而且我还添加了多个 `When` 和随后的 `Then` 子句，进一步描绘“My Lists”页面。

E.4 编写步骤函数

下面编写 Gherkin 句法描述的特性对应的“步骤”函数，即真正使用代码实现。

生成占位步骤

运行 `behave`，它会告诉我们需要实现的每一步：

```
$ python manage.py behave
Feature: My Lists # features/my_lists.feature:1
  As a logged-in user
  I want to be able to see all my lists in one page
  So that I can find them all after I've written them
  Scenario: Create two lists and see them on the My Lists page #
features/my_lists.feature:6
    Given I am a logged-in user                                 # None
    Given I am a logged-in user                                 # None
    When I create a list with first item "Reticulate Splines"  # None
    And I add an item "Immanentize Eschaton"                    # None
    And I create a list with first item "Buy milk"              # None
    Then I will see a link to "My lists"                        # None
    When I click the link to "My lists"                         # None
    Then I will see a link to "Reticulate Splines"              # None
    And I will see a link to "Buy milk"                         # None
    When I click the link to "Reticulate Splines"               # None
    Then I will be on the "Reticulate Splines" list page        # None


Failing scenarios:
  features/my_lists.feature:6 Create two lists and see them on the My Lists
page


0 features passed, 1 failed, 0 skipped
0 scenarios passed, 1 failed, 0 skipped
0 steps passed, 0 failed, 0 skipped, 10 undefined
Took 0m0.000s

You can implement step definitions for undefined steps with these snippets:

@given(u'I am a logged-in user')
def step_impl(context):
    raise NotImplementedError(u'STEP: Given I am a logged-in user')

@when(u'I create a list with first item "Reticulate Splines"')
def step_impl(context):
[...]
```

而且你会发现，输出有不同的颜色，如图 E-1 所示。

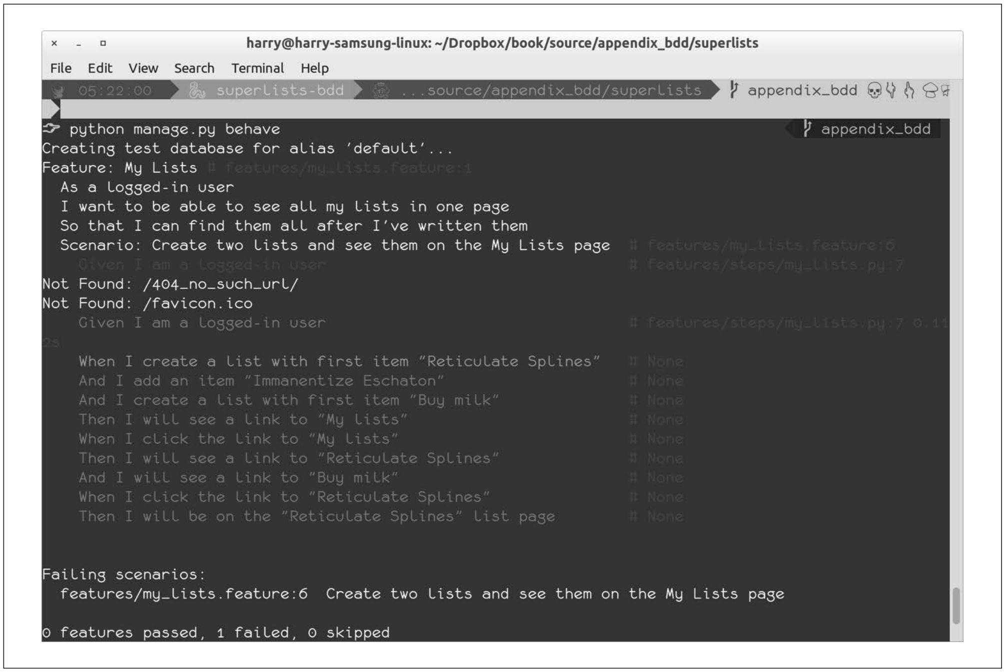


图 E-1：Behave 在控制台输出带有不同颜色的内容

Behave 建议我们复制粘贴这些片段，在此基础上构建步骤。

E.5 定义第一步

下面尝试实现“Given I am a logged-in user”这一步。我直接从 functional_tests/test_my_lists.py 中复制 `self.create_pre_authenticated_session` 的代码，然后稍做调整（例如删掉服务器端版本，不过后面再加上也容易）。

features/steps/my_lists.py

```
from behave import given, when, then
from functional_tests.management.commands.create_session import \
    create_pre_authenticated_session
from django.conf import settings


@given('I am a logged-in user')
def given_i_am_logged_in(context):
    session_key = create_pre_authenticated_session(email='edith@example.com')
    ## 为了设定cookie，我们要先访问网站
    ## 而404页面是加载最快的
    context.browser.get(context.get_url("/404_no_such_url/"))
    context.browser.add_cookie(dict(
```

```
        name=settings.SESSION_COOKIE_NAME,
        value=session_key,
        path='/',
    ))
```

context 变量需要稍做说明：它有点像是全局变量，因为我们将通过它存储在步骤之间共享的信息，把它传给要执行的每一步。这里，我们假定它存储了一个浏览器对象和 server_url。我们将多次使用这个变量，就像在使用 unittest 编写功能测试时经常使用 self 一样。

E.6　environment.py中与setUp和tearDown等价的函数

各步可以修改 context 中的状态，不过前期准备工作，即与 setUp 等价的操作，在 environment.py 文件中设置：

features/environment.py

```
from selenium import webdriver

def before_all(context):
    context.browser = webdriver.Firefox()

def after_all(context):
    context.browser.quit()

def before_feature(context, feature):
    pass
```

E.7　再次运行

可以再运行一次，做健全性检查，确认新编写的步骤是否可行，以及能否启动浏览器：

```
$ python manage.py behave
[...]
1 step passed, 0 failed, 0 skipped, 9 undefined
```

输出的内容很多，不过可以看到，第一步通过了。下面定义余下的步骤。

E.8　在步骤中捕获参数

我们将说明如何在步骤描述中捕获参数。接下来的一步是：

features/my_lists.feature

```
    When I create a list with first item "Reticulate Splines"
```

自动生成的步骤定义如下：

features/steps/my_lists.py

```
@given('I create a list with first item "Reticulate Splines"')
def step_impl(context):
    raise NotImplementedError(
        u'STEP: When I create a list with first item "Reticulate Splines"'
    )
```

我们希望以任意的第一个待办事项创建清单。所以，如果能通过某种方式捕获双引号中的内容就好了，这样就可以把它作为参数传给更为通用的函数。这是 BDD 的一个常见需求，Behave 为此提供了优雅的句法。还记得 Python 为字符串格式化提供的新句法吗？

features/steps/my_lists.py (ch35l006)

```
[...]

@when('I create a list with first item "{first_item_text}"')
def create_a_list(context, first_item_text):
    context.browser.get(context.get_url('/'))
    context.browser.find_element_by_id('id_text').send_keys(first_item_text)
    context.browser.find_element_by_id('id_text').send_keys(Keys.ENTER)
    wait_for_list_item(context, first_item_text)
```

很棒吧？

在步骤中捕获参数是 BDD 句法最为强大的功能之一。

与在 Selenium 测试中一样，我们要显式等待。依旧使用 base.py 中的 `@wait` 装饰器：

features/steps/my_lists.py (ch35l007)

```
from functional_tests.base import wait
[...]


@wait
def wait_for_list_item(context, item_text):
    context.test.assertIn(
        item_text,
        context.browser.find_element_by_css_selector('#id_list_table').text
    )
```

与之类似，我们也可以把待办事项添加到现有清单中，查看或点击链接：

features/steps/my_lists.py (ch35l008)

```
from selenium.webdriver.common.keys import Keys
[...]


@when('I add an item "{item_text}"')
def add_an_item(context, item_text):
```

```
    context.browser.find_element_by_id('id_text').send_keys(item_text)
    context.browser.find_element_by_id('id_text').send_keys(Keys.ENTER)
    wait_for_list_item(context, item_text)


@then('I will see a link to "{link_text}"')
@wait
def see_a_link(context, link_text):
    context.browser.find_element_by_link_text(link_text)


@when('I click the link to "{link_text}"')
def click_link(context, link_text):
    context.browser.find_element_by_link_text(link_text).click()
```

注意，我们甚至可以在步骤上使用 `@wait` 装饰器。

最后是稍微复杂一些的步骤，描述自己在某个清单的页面上：

features/steps/my_lists.py (ch35l009)

```
@then('I will be on the "{first_item_text}" list page')
@wait
def on_list_page(context, first_item_text):
    first_row = context.browser.find_element_by_css_selector(
        '#id_list_table tr:first-child'
    )
    expected_row_text = '1: ' + first_item_text
    context.test.assertEqual(first_row.text, expected_row_text)
```

现在运行，得到第一个预期失败：

```
$ python manage.py behave

Feature: My Lists # features/my_lists.feature:1
  As a logged-in user
  I want to be able to see all my lists in one page
  So that I can find them all after I've written them
  Scenario: Create two lists and see them on the My Lists page #
features/my_lists.feature:6
    Given I am a logged-in user                                  #
features/steps/my_lists.py:19
    When I create a list with first item "Reticulate Splines"  #
features/steps/my_lists.py:31
    And I add an item "Immanentize Eschaton"                     #
features/steps/my_lists.py:39
    And I create a list with first item "Buy milk"               #
features/steps/my_lists.py:31
    Then I will see a link to "My lists"                         #
functional_tests/base.py:12
      Traceback (most recent call last):
[...]
        File "features/steps/my_lists.py", line 49, in see_a_link
          context.browser.find_element_by_link_text(link_text)
[...]
```

```
    selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: My lists

[...]

Failing scenarios:
  features/my_lists.feature:6  Create two lists and see them on the My Lists
page

0 features passed, 1 failed, 0 skipped
0 scenarios passed, 1 failed, 0 skipped
4 steps passed, 1 failed, 5 skipped, 0 undefined
```

从输出可以看出，我们在“用户故事”上走了多远：我们成功创建了两个清单，但是“My Lists”链接未出现。

E.9 与行间式功能测试比较

我不会完整实现整个功能，不过你可以看出，这与行间式功能测试一样，能驱动我们向前开发。

下面回顾一下行间测试，比较一下：

functional_tests/test_my_lists.py

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    # 伊迪丝是已登录用户
    self.create_pre_authenticated_session('edith@example.com')

    # 她访问首页，新建一个清单
    self.browser.get(self.live_server_url)
    self.add_list_item('Reticulate splines')
    self.add_list_item('Immanentize eschaton')
    first_list_url = self.browser.current_url

    # 她第一次看到“My Lists”链接
    self.browser.find_element_by_link_text('My lists').click()

    # 她看到这个页面中有她创建的清单
    # 而且清单根据第一个待办事项命名
    self.wait_for(
        lambda: self.browser.find_element_by_link_text('Reticulate splines')
    )
    self.browser.find_element_by_link_text('Reticulate splines').click()
    self.wait_for(
        lambda: self.assertEqual(self.browser.current_url, first_list_url)
    )

    # 她决定再建一个清单试试
    self.browser.get(self.live_server_url)
    self.add_list_item('Click cows')
    second_list_url = self.browser.current_url
```

```
        # 这个新建的清单也在“My Lists”页面显示出来了
        self.browser.find_element_by_link_text('My lists').click()
        self.wait_for(
            lambda: self.browser.find_element_by_link_text('Click cows')
        )
        self.browser.find_element_by_link_text('Click cows').click()
        self.wait_for(
            lambda: self.assertEqual(self.browser.current_url, second_list_url)
        )

        # 她退出后，“My Lists”链接不见了
        self.browser.find_element_by_link_text('Log out').click()
        self.wait_for(lambda: self.assertEqual(
            self.browser.find_elements_by_link_text('My lists'),
            []
        ))
```

虽然不能一一对应比较，但是可以看看代码行数，见表 E-1。

表E-1：比较代码行数

BDD	标准的功能测试
特性描述文件：20（3 个可选）	测试函数的主体：45
步骤文件：56 行	辅助函数：23

这样比较并不严谨，但是可以认为特性描述文件和“标准的功能测试”的测试函数主体是等价的，都表示测试的主体“故事”，而步骤定义和辅助函数表示“隐藏的”实现细节。如果把行数加在一起，总行数相差不多，但是二者的结果不一样：BDD 测试写出的故事更简洁，而且更多内容隐藏到实现细节中了。

E.10　BDD得到的测试代码结构更好

对我而言，真正吸引人的是，BDD 工具迫使我们思考测试代码的结构。在行间式功能测试中，实现需要多少行代码就可以写多少行，用户故事是通过注释描述的。我们很难控制自己不从别处或同一个测试的前面复制粘贴代码。到目前为止，你可以看到，我只定义了几个辅助函数（例如 `get_item_input_box`）。

与之相比，BDD 句法则强制我们为每一步编写单独的函数，因此很多代码都是可以重用的。

- 新建清单。
- 把待办事项添加到现有清单中。
- 点击特定文本的链接。
- 断言我在查看某个清单的页面。

通过 BDD 写出的代码与业务逻辑匹配得更好，而且能分层抽象，把功能测试的故事与实现的代码分开。

这样做最终的好处是，如果想更换编程语言，理论上可以原样保留 Gherkin 句法编写的特性描述，丢掉 Python 代码实现的步骤，再用新语言重新编写。

E.11 与页面模式比较

第 25 章举了一个使用“页面模式”的示例。页面模式是组织 Selenium 测试的面向对象方式。下面回顾一下它的用法：

functional_tests/test_sharing.py

```
from .lists_page import ListsPage
[...]

class SharingTest(FunctionalTest):

    def test_can_share_a_list_with_another_user(self):
        # [...]
        self.browser.get(self.live_server_url)
        list_page = ListPage(self).add_list_item('Get help')

        # 她看到“分享这个清单”选项
        share_box = list_page.get_share_box()
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )

        # 她分享自己的清单之后，页面更新了
        # 提示已经分享给Oniciferous
        list_page.share_list_with('oniciferous@example.com')
```

Page 类的定义如下：

functional_tests/lists_pages.py

```
class ListPage(object):

    def __init__(self, test):
        self.test = test


    def get_table_rows(self):
        return self.test.browser.find_elements_by_css_selector('#id_list_table tr')


    @wait
    def wait_for_row_in_list_table(self, item_text, item_number):
        row_text = '{}: {}'.format(item_number, item_text)
        rows = self.get_table_rows()
        self.test.assertIn(row_text, [row.text for row in rows])


    def get_item_input_box(self):
        return self.test.browser.find_element_by_id('id_text')
```

可以看出，不管是使用页面模式还是其他结构，完全可以在行间式功能测试中做同样的抽象，实现某种 DSL。但是，我们应该自律，而不能靠框架去约束。

其实在 BDD 中也可以使用页面模式，在实现步骤时通过它浏览网站中的页面。

E.12　BDD可能没有行间注释的表达力强

另一方面，我觉得 Gherkin 句法有点不够灵活。行间式注释表达力强，而且可读性高，但 BDD 的特性描述有点拗口：

functional_tests/test_my_lists.py

```
        # 伊迪丝是已登录用户
        # 她访问首页，新建一个清单
        # 她第一次看到“My Lists”链接
        # 她看到这个页面中有她创建的清单
        # 而且清单根据第一个待办事项命名
        # 她决定再建一个清单试试
        # 这个新建的清单在“My Lists”页面也显示出来了
        # 她退出后，“My Lists”链接不见了
    [...]
```

与呆板的“Given/Then/When”结构相比，功能测试中的行间注释可读性更高，也显得更自然，而且在一定程度上，更能从用户的角度思考问题。（Gherkin 也支持在特性描述文件中编写“注释”，这能在一定程度上缓解上述问题，但是我想用的人并不多。）

E.13　非程序员会编写测试吗

有一点我还没有提到：BDD 最初的动机之一是让非程序员（可能是业务代表或客户代表）能够编写 Gherkin 句法。我十分怀疑现实中有没有人这么做；即便有，与 BDD 的其他优势相比，我想这也不算什么。

E.14　目前的结论

我才刚接触 BDD，还不能得出什么强有力的结论。我觉得“强制”把功能测试分成不同的步骤十分吸引人，因为这样便于在功能测试中大量重用代码，而且能把关注点明确分开，一边是对故事的描述，另一边是具体实现。此外，BDD 还能让我们站在业务逻辑的角度思考问题，而不是想着“应该如何使用 Selenium 去做”。

但是，世界上没有免费的午餐。与功能测试中行间注释的无拘无束相比，Gherkin 句法太死板、不够灵活。

我还想知道，当功能描述由一两个变成十几个、步骤定义由四五个变成几百行代码之后，BDD 能否适应。

总之，我觉得 BDD 绝对值得深入研究，我的下一个个人项目可能会使用它。

感谢 Daniel Pope、Rachel Willmer 和 Jared Contrascere 对本附录的反馈。

BDD 总结

- 有助于编写结构良好、可重用的测试代码
  BDD 能分离关注点，把功能测试拆分成人类可读的“特性描述”文件（使用 Gherkin 句法）和步骤函数的实现，这样有助于编写可重用、易于管理的测试代码。
- 可能有失可读性
  Gherkin 句法虽然追求的是人类可读性，但是并没有充分发挥人类语言的灵活性，因此可能无法像行间注释那样注重细节、明确表明意图。
- 多尝试总是好的
  我不断强调，我还未在真实的项目中用过 BDD，因此你要对我讲的内容持怀疑态度。但是我强烈推荐你使用 BDD。我将试着在我的下一个项目中使用它，同时也建议你这么做。

附录F

构建一个REST API：JSON、Ajax和JavaScript模拟技术

表现层状态转化（REpresentational State Transfer，REST）是一种设计 Web 服务的方案，读取和更新的是“资源”（resource）。设计通过 Web 使用的 API 时，这是首选方案。

我们设计的 Web 应用还用不到 API，那为什么现在就要设计一个呢？其中一个动机是想让网站更加动态，从而提升用户体验。在清单中添加待办事项后，我们不想等待页面刷新，而是想使用 JavaScript 向 API 发送异步请求，让用户感受网站的交互性。

但更重要的一点或许是，有了 API，我们就可以通过浏览器之外的机制与后端应用交互。例如，客户端可以是移动应用，也可以是命令行应用，而且其他开发者还可以围绕后端构建库和工具。

本附录将说明如何自己动手构建一个 API。附录 G 再介绍 Django 生态系统中的一个流行工具——Django-Rest-Framework。

F.1　本附录采用的方案

我们构建的 API 不涵盖应用的全部功能，而是假设已经有清单了。根据 REST 架构，URL 和 HTTP 方法（常用的有 GET 和 POST，不过也有比较少用的，例如 PUT 和 DELETE）之间有一定的对应关系，我们可以据此设计方案。

维基百科中的 REST 词条对此有很好的概述，简单来说：

- 新 URL 结构为 /api/lists/{id}/；
- 通过 GET 请求获取清单详情（包括清单中的全部待办事项）的 JSON 格式；
- 通过 POST 请求添加待办事项。

我们将使用第 25 章结束时的代码。

F.2　选择测试方案

如果构建的 API 对客户端一无所知，或许应该好好想想测试应该下行到哪一层。对功能测试来说，我们仍然要启动一个真实的服务器（可以使用 `LiveServerTestCase`），然后使用 `requests` 库与之交互。我们应该仔细考虑如何设置固件（如果使用 API 自身，测试之间牵涉的依赖太多），以及哪一层的单元测试对我们最有用。或者，干脆只使用 Django 测试客户端做一层测试。

现在的情况是，我们要为基于浏览器的客户端构建 API。我们想在线上网站使用这个 API，而且对应用之前的功能没有任何影响。因此，我们依然要让功能测试做最高层的测试，通过功能测试检查 JavaScript 和 API 之间的集成情况。

这样一来，低层测试就要使用 Django 测试客户端了。下面开始构建。

F.3　基本结构

首先，编写一个功能测试，检查新的 URL 结构能正常响应 GET 请求（状态码为 200），而且响应是 JSON 格式（而不是 HTML 格式）：

lists/tests/test_api.py

```
import json
from django.test import TestCase

from lists.models import List, Item


class ListAPITest(TestCase):
    base_url = '/api/lists/{}/'   ❶

    def test_get_returns_json_200(self):
        list_ = List.objects.create()
        response = self.client.get(self.base_url.format(list_.id))
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response['content-type'], 'application/json')
```

❶ 使用类级常量指定 URL 是本附录采用的新方式，这样做便无须重复硬编码 URL。此外，还可以调用 `reverse`，进一步减少重复。

然后，引入 urls 文件：

superlists/urls.py

```
from django.conf.urls import include, url
from accounts import urls as accounts_urls
from lists import views as list_views
from lists import api_urls
from lists import urls as list_urls

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)),
    url(r'^accounts/', include(accounts_urls)),
    url(r'^api/', include(api_urls)),
]
```

和：

lists/api_urls.py

```
from django.conf.urls import url
from lists import api

urlpatterns = [
    url(r'^lists/(\d+)/$', api.list, name='api_list'),
]
```

API 的核心代码可以放在 api.py 文件中，只需三行代码就行了：

lists/api.py

```
from django.http import HttpResponse

def list(request, list_id):
    return HttpResponse(content_type='application/json')
```

测试应该能通过。现在就有了基础结构。

```
$ python manage.py test lists
[...]
..................................................
 ---------------------------------------------------------------------
Ran 50 tests in 0.177s

OK
```

F.4　返回实质内容

接下来，我们要让 API 返回一些实质内容，即清单中各待办事项的 JSON 表示形式：

lists/tests/test_api.py (ch36l002)

```
    def test_get_returns_items_for_correct_list(self):
        other_list = List.objects.create()
        Item.objects.create(list=other_list, text='item 1')
        our_list = List.objects.create()
```

```
        item1 = Item.objects.create(list=our_list, text='item 1')
        item2 = Item.objects.create(list=our_list, text='item 2')
        response = self.client.get(self.base_url.format(our_list.id))
        self.assertEqual(
            json.loads(response.content.decode('utf8')),  ❶
            [
                {'id': item1.id, 'text': item1.text},
                {'id': item2.id, 'text': item2.text},
            ]
        )
```

❶ 这个测试主要要注意这一点。我们期望响应是 JSON 格式，使用 `json.loads()` 是因为测试 Python 对象比直接处理原始的 JSON 字符串容易。

实现时则要反过来，使用 `json.dumps()`：

lists/api.py

```
import json
from django.http import HttpResponse
from lists.models import List, Item


def list(request, list_id):
    list_ = List.objects.get(id=list_id)
    item_dicts = [
        {'id': item.id, 'text': item.text}
        for item in list_.item_set.all()
    ]
    return HttpResponse(
        json.dumps(item_dicts),
        content_type='application/json'
    )
```

这是使用列表推导的好机会！

F.5 添加对POST请求的支持

这个 API 还要允许通过 POST 请求向清单中添加新待办事项。先采用常规方式：

lists/tests/test_api.py (ch36l004)

```
    def test_POSTing_a_new_item(self):
        list_ = List.objects.create()
        response = self.client.post(
            self.base_url.format(list_.id),
            {'text': 'new item'},
        )
        self.assertEqual(response.status_code, 201)
        new_item = list_.item_set.get()
        self.assertEqual(new_item.text, 'new item')
```

实现同样简单，基本上与常规的视图所做的一样，不过这里要返回 201 响应，而不能重定向：

lists/api.py (ch36l005)

```
def list(request, list_id):
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
        Item.objects.create(list=list_, text=request.POST['text'])
        return HttpResponse(status=201)
    item_dicts = [
        [...]
```

这样便可以了：

```
$ python manage.py test lists
[...]

Ran 52 tests in 0.177s

OK
```

> 构建 REST API 的要点之一是，要充分利用 HTTP 状态码。

F.6　使用Sinon.js测试客户端Ajax

没有模拟库是**无法**测试 Ajax 的。不同的测试框架和工具采用不同的模拟库，而 Sinon 是通用的。稍后你将看到，Sinon 还提供了 JavaScript 驭件。

下载 Sinon，将其放到 lists/static/tests/ 文件夹中。

下面编写第一个 Ajax 测试：

lists/static/tests/tests.html (ch36l007)

```
  <div id="qunit-fixture">
    <form>
      <input name="text" />
      <div class="has-error">Error text</div>
    </form>
    <table id="id_list_table">  ❶
    </table>
  </div>

  <script src="../jquery-3.1.1.min.js"></script>
  <script src="../list.js"></script>
  <script src="qunit-2.0.1.js"></script>
  <script src="sinon-1.17.6.js"></script>  ❷

  <script>
/* global sinon */

var server;
QUnit.testStart(function () {
```

```
  server = sinon.fakeServer.create();  ❸
});
QUnit.testDone(function () {
  server.restore();  ❸
});

QUnit.test("errors should be hidden on keypress", function (assert) {
[...]


QUnit.test("should get items by ajax on initialize", function (assert) {
  var url = '/getitems/';
  window.Superlists.initialize(url);

  assert.equal(server.requests.length, 1);  ❹
  var request = server.requests[0];
  assert.equal(request.url, url);
  assert.equal(request.method, 'GET');
});

  </script>
```

❶ 在固件 `div` 元素中添加一个元素，表示清单表格。

❷ 导入 sinon.js（要先下载，并放到正确的文件夹中）。

❸ QUnit 中的 `testStart` 和 `testDone` 对应于 Python 测试中的 `setUp` 和 `tearDown`。这里，我们让 Sinon 启动 Ajax 测试工具（`fakeServer`），并将它赋值给全局作用域中的 `server` 变量。

❹ 然后通过 `server` 对代码发送的 Ajax 请求下断言。这里，我们测试请求的目标 URL 和所用的 HTTP 方法。

为了发送 Ajax 请求，我们将使用 jQuery 提供的 Ajax 辅助方法，这比使用浏览器底层的标准 `XMLHttpRequest` 对象**要简单得多**：

lists/static/list.js

```
@@ -1,6 +1,10 @@
 window.Superlists = {};
-window.Superlists.initialize = function () {
+window.Superlists.initialize = function (url) {
   $('input[name="text"]').on('keypress', function () {
     $('.has-error').hide();
   });
+
+  $.get(url);
+
 };
+
```

现在测试应该能通过：

```
5 assertions of 5 passed, 0 failed.
1. errors should be hidden on keypress (1)
```

```
2. errors aren't hidden if there is no keypress (1)
3. should get items by ajax on initialize (3)
```

好吧，我们能向服务器发送 GET 请求了，但是**具体的操作**呢？我们应该如何测试“异步”请求，应该如何处理（最终得到的）响应呢？

使用Sinon测试Ajax请求的异步行为

这是人们喜欢 Sinon 的主要原因。我们可以通过 `server.respond()` 准确控制异步代码的流程：

lists/static/tests/tests.html (ch36l009)

```
QUnit.test("should fill in lists table from ajax response", function (assert) {
  var url = '/getitems/';
  var responseData = [
    {'id': 101, 'text': 'item 1 text'},
    {'id': 102, 'text': 'item 2 text'},
  ];
  server.respondWith('GET', url, [
    200, {"Content-Type": "application/json"}, JSON.stringify(responseData)  ❶
  ]);
  window.Superlists.initialize(url);  ❷

  server.respond();  ❸

  var rows = $('#id_list_table tr');  ❹
  assert.equal(rows.length, 2);
  var row1 = $('#id_list_table tr:first-child td');
  assert.equal(row1.text(), '1: item 1 text');
  var row2 = $('#id_list_table tr:last-child td');
  assert.equal(row2.text(), '2: item 2 text');
});
```

❶ 设定一些响应数据供 Sinon 使用。我们设定了状态码、首部以及希望服务器返回的 JSON 响应——这是最重要的。

❷ 然后调用要测试的函数。

❸ 关键时刻到了。随后，我们可以在任何需要的地方调用 `server.respond()`，触发 Ajax 循环中所有的异步代码，即用于处理响应的那些回调。

❹ 然后检查 Ajax 回调有没有在表格的行中填充新的待办事项。

实现如下所示：

lists/static/list.js (ch36l010)

```
  if (url) {
    $.get(url).done(function (response) {
      var rows = '';
      for (var i=0; i<response.length; i++) {
        var item = response[i];
        rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
      }
```

```
    $('#id_list_table').html(rows);
  });
}
```

我们很幸运，因为 jQuery 是使用 `.done()` 函数为 Ajax 注册回调的。如果使用标准的 `JavaScript Promise .then()` 回调，异步操作就要多一层。不过，QUnit 也能处理，详情参见 `async` 函数的文档。其他测试框架为此也提供了类似的函数。

F.7 在模板中连接各部分，确认这样是否真的可行

我们先做个破坏，把 lists.html 模板中用于显示清单表格的 `{% for %}` 循环删掉：

lists/templates/list.html

```
@@ -6,9 +6,6 @@

 {% block table %}
   <table id="id_list_table" class="table">
-    {% for item in list.item_set.all %}
-      <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
-    {% endfor %}
   </table>

   {% if list.owner %}
```

这会导致其中一个单元测试失败，可以先暂时删除那个测试。

优雅降级和渐进增强

删除非 Ajax 版本的清单页面之后，就无法优雅降级了，即没有在禁用 JavaScript 的情况下依然能正常使用的版本。

以前，优雅降级通常是为了提供辅助功能，因为供视觉障碍人士使用的“屏幕阅读器”通常不支持 JavaScript，所以完全依赖 JavaScript 就把这部分用户排除在外了。但据我了解，现在这已经不是什么大问题了。但有些用户甚至会基于安全方面的原因而禁用 JavaScript。

另一个常见的问题是，为不同的浏览器提供不同程度的 JavaScript 支持。当你开始迈向“现代的”前端开发和 ES2015 时，尤其要注意这个问题。

简单来说，最好始终提供非 JavaScript 版本的“后援”。如果已经先构建好了无须 JavaScript 就能正常使用的网站，就千万不要轻易把“陈旧的”HTML 版本删除。我这么做只是因为删除后更便于说明我想讲的内容。

这样做会导致功能测试失败：

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
FAIL: test_can_start_a_list_for_one_user
[...]
  File "/.../superlists/functional_tests/test_simple_list_creation.py", line
32, in test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('1: Buy peacock feathers')
[...]
AssertionError: '1: Buy peacock feathers' not found in []
[...]
FAIL: test_multiple_users_can_start_lists_at_different_urls

FAILED (failures=2)
```

下面在基模板中添加一个 `{% scripts %}` 块，这样便可以在清单页面有选择地覆盖：

lists/templates/base.html

```
    <script src="/static/list.js"></script>

    {% block scripts %}
      <script>
$(document).ready(function () {
  window.Superlists.initialize();
});
      </script>
    {% endblock scripts %}

  </body>
```

然后，在 list.html 中稍微修改一下调用 `initialize` 的方式，传入正确的 URL：

lists/templates/list.html (ch36l016)

```
{% block scripts %}
  <script>
$(document).ready(function () {
  var url = "{% url 'api_list' list.id %}";
  window.Superlists.initialize(url);
});
  </script>
{% endblock scripts %}
```

你猜怎么着？测试通过了！

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
Ran 2 test in 11.730s

OK
```

这是个不错的开始！

如果这时运行功能测试，你会发现其他功能测试中有几个失败。接下来我们就要处理这些

失败。此外，这里还在使用通过表单处理 POST 请求的过时方法，页面需要刷新，离时下流行的单页应用还有段距离。但我们正向着目标前进！

F.8 实现Ajax POST，包括CSRF令牌

首先为清单表格设定一个 `id`，以便在 JavaScript 代码中引用它：

lists/templates/base.html

```
<h1>{% block header_text %}{% endblock %}</h1>
{% block list_form %}
  <form id="id_item_form" method="POST" action="{% block form_action %}{% endblock %}">
    {{ form.text }}
     [...]
```

然后使用 ID 调整 JavaScript 测试中的固件，并加上页面当前的 CSRF 令牌：

lists/static/tests/tests.html

```
@@ -9,9 +9,14 @@
 <body>
   <div id="qunit"></div>
   <div id="qunit-fixture">
-    <form>
+    <form id="id_item_form">
       <input name="text" />
-      <div class="has-error">Error text</div>
+      <input type="hidden" name="csrfmiddlewaretoken" value="tokey" />
+      <div class="has-error">
+        <div class="help-block">
+          Error text
+        </div>
+      </div>
     </form>
```

测试如下：

lists/static/tests/tests.html (ch36l019)

```
QUnit.test("should intercept form submit and do ajax post", function (assert) {
  var url = '/listitemsapi/';
  window.Superlists.initialize(url);

  $('#id_item_form input[name="text"]').val('user input');  ❶
  $('#id_item_form input[name="csrfmiddlewaretoken"]').val('tokeney');  ❶
  $('#id_item_form').submit();  ❶

  assert.equal(server.requests.length, 2);  ❷
  var request = server.requests[1];
  assert.equal(request.url, url);
  assert.equal(request.method, "POST");
  assert.equal(
    request.requestBody,
    'text=user+input&csrfmiddlewaretoken=tokeney'  ❸
  );
});
```

❶ 模拟用户操作，填表后点击提交按钮。

❷ 预期会有第二个 Ajax 请求（第一个是针对清单表格的 GET 请求）。

❸ 检查 POST 请求的 `requestBody`。可以看出，它的值经过 URL 编码了，这虽然不是最易于测试的，但是可读性尚可。

实现方式如下：

lists/static/list.js

```
[...]
  $('#id_list_table').html(rows);
});

var form = $('#id_item_form');
form.on('submit', function(event) {
  event.preventDefault();
  $.post(url, {
    'text': form.find('input[name="text"]').val(),
    'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
  });
});
```

现在 JavaScript 测试能通过了，但是功能测试却失败了，因为虽然 POST 请求成功了，但是却没有更新页面，显示新添加的待办事项：

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

F.9 JavaScript中的模拟技术

我们希望处理完 Ajax POST 请求后，客户端能更新表格中的待办事项。其实这与重新加载页面所做的事情是一样的，我们要从服务器中获取清单中当前的待办事项，然后在表格中显示出来。

看来，我们需要一个辅助函数。

lists/static/list.js

```
window.Superlists = {};

window.Superlists.updateItems = function (url) {
  $.get(url).done(function (response) {
    var rows = '';
    for (var i=0; i<response.length; i++) {
      var item = response[i];
      rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
    }
    $('#id_list_table').html(rows);
  });
```

```
};

window.Superlists.initialize = function (url) {
  $('input[name="text"]').on('keypress', function () {
    $('.has-error').hide();
  });

  if (url) {
    window.Superlists.updateItems(url);

    var form = $('#id_item_form');
    [...]
```

这只算是一次重构。经确认，现在 JavaScript 测试依然能通过：

```
12 assertions of 12 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should get items by ajax on initialize (3)
4. should fill in lists table from ajax response (3)
5. should intercept form submit and do ajax post (4)
```

那么，我们应该如何测试 Ajax POST 请求成功后调用了 `updateItems` 呢？我们可不想傻傻地重复编写模拟服务器响应的代码，然后自己动手检查待办事项表格……要不使用驭件试试？

首先，设置一个“沙盒”，让它跟踪我们创建的所有驭件，并在每次测试之后把被模拟的东西还原。

lists/static/tests/tests.html (ch36l023)

```
var server, sandbox;
QUnit.testStart(function () {
  server = sinon.fakeServer.create();
  sandbox = sinon.sandbox.create();
});
QUnit.testDone(function () {
  server.restore();
  sandbox.restore();  ❶
});
```

❶ `.restore()` 是重点，它的作用是在每次测试之后还原被模拟的东西。

lists/static/tests/tests.html (ch36l024)

```
QUnit.test("should call updateItems after successful post", function (assert) {
  var url = '/listitemsapi/';
  window.Superlists.initialize(url);  ❶
  var response = [
    201,
    {"Content-Type": "application/json"},
    JSON.stringify({}),
  ];
  server.respondWith('POST', url, response);  ❶
  $('#id_item_form input[name="text"]').val('user input');
```

```
  $('#id_item_form input[name="csrfmiddlewaretoken"]').val('tokeney');
  $('#id_item_form').submit();

  sandbox.spy(window.Superlists, 'updateItems');  ❷
  server.respond();  ❷

  assert.equal(
    window.Superlists.updateItems.lastCall.args,  ❸
    url
  );
});
```

❶ 首先要注意，初始化**之后**才能设置服务器响应。这是因为我们想设置的是提交表单时发送的 POST 请求的响应，而不是一开始那个 GET 请求的响应。（还记得第 16 章所学的知识吗？JavaScript 测试最难掌握的技术之一便是控制执行顺序。）

❷ 同样，仅当开始那个 GET 请求处理完毕**之后**，我们才开始模拟辅助函数。`sandbox.spy` 调用的作用与 Python 测试中的 `patch` 一样，把指定对象替换为驭件。

❸ 模拟的 `updateItems` 函数现在多了一些属性，例如 `lastCall` 和 `lastCall.args`（类似于 Python 驭件的 `call_args`）。

让测试通过之前，我们想故意犯个错，确认它的确能测试我们想测试的行为：

lists/static/list.js

```
$.post(url, {
  'text': form.find('input[name="text"]').val(),
  'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
}).done(function () {
  window.Superlists.updateItems();
});
```

收效不错，但功能测试还未全部通过：

```
12 assertions of 13 passed, 1 failed.
[...]
6. should call updateItems after successful post (1, 0, 1)
    1. failed
        Expected: "/listitemsapi/"
        Result: []
        Diff: "/listitemsapi/"[]
        Source: file:///.../superlists/lists/static/tests/tests.html:124:15
```

据此修正：

lists/static/list.js

```
    }).done(function () {
      window.Superlists.updateItems(url);
    });
```

现在功能测试通过了！或者，至少部分通过了。其他测试还有问题，稍后再回来解决。

结束重构：让测试与代码匹配

现在，我有点不舒心，因为重构还没结束。下面稍微让单元测试与代码匹配一些：

lists/static/tests/tests.html

```
@@ -50,9 +50,19 @@ QUnit.testDone(function () {
 });


-QUnit.test("should get items by ajax on initialize", function (assert) {
+QUnit.test("should call updateItems on initialize", function (assert) {
   var url = '/getitems/';
+  sandbox.spy(window.Superlists, 'updateItems');
   window.Superlists.initialize(url);
+  assert.equal(
+    window.Superlists.updateItems.lastCall.args,
+    url
+  );
+});
+
+QUnit.test("updateItems should get correct url by ajax", function (assert) {
+  var url = '/getitems/';
+  window.Superlists.updateItems(url);

   assert.equal(server.requests.length, 1);
   var request = server.requests[0];
@@ -60,7 +70,7 @@ QUnit.test("should get items by ajax on initialize", function (assert) {
   assert.equal(request.method, 'GET');
 });

-QUnit.test("should fill in lists table from ajax response", function (assert) {
+QUnit.test("updateItems should fill in lists table from ajax response", function (assert) {
   var url = '/getitems/';
   var responseData = [
     {'id': 101, 'text': 'item 1 text'},
@@ -69,7 +79,7 @@ QUnit.test("should fill in lists table from ajax response", function [...]
   server.respondWith('GET', url, [
     200, {"Content-Type": "application/json"}, JSON.stringify(responseData)
   ]);
-  window.Superlists.initialize(url);
+  window.Superlists.updateItems(url);

   server.respond();
```

现在测试的结果如下：

```
14 assertions of 14 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should call updateItems on initialize (1)
4. updateItems should get correct url by ajax (3)
5. updateItems should fill in lists table from ajax response (3)
6. should intercept form submit and do ajax post (4)
7. should call updateItems after successful post (1)
```

F.10 数据验证：留给读者的练习

如果运行全部测试，你会发现有两个针对验证的功能测试失败了：

```
$ python manage.py test
[...]
ERROR: test_cannot_add_duplicate_items
(functional_tests.test_list_item_validation.ItemValidationTest)
[...]
ERROR: test_error_messages_are_cleared_on_input
(functional_tests.test_list_item_validation.ItemValidationTest)
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .has-error
```

我不会告诉你具体应该怎么解决，下面仅给出所需的单元测试：

lists/tests/test_api.py (ch36l027)

```
from lists.forms import DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR
[...]
    def post_empty_input(self):
        list_ = List.objects.create()
        return self.client.post(
            self.base_url.format(list_.id),
            data={'text': ''}
        )


    def test_for_invalid_input_nothing_saved_to_db(self):
        self.post_empty_input()
        self.assertEqual(Item.objects.count(), 0)


    def test_for_invalid_input_returns_error_code(self):
        response = self.post_empty_input()
        self.assertEqual(response.status_code, 400)
        self.assertEqual(
            json.loads(response.content.decode('utf8')),
            {'error': EMPTY_ITEM_ERROR}
        )


    def test_duplicate_items_error(self):
        list_ = List.objects.create()
        self.client.post(
            self.base_url.format(list_.id), data={'text': 'thing'}
        )
        response = self.client.post(
            self.base_url.format(list_.id), data={'text': 'thing'}
        )
        self.assertEqual(response.status_code, 400)
        self.assertEqual(
            json.loads(response.content.decode('utf8')),
```

```
            {'error': DUPLICATE_ITEM_ERROR}
        )
```

以及 JavaScript 测试：

lists/static/tests/tests.html (ch36l029-2)

```
QUnit.test("should display errors on post failure", function (assert) {
  var url = '/listitemsapi/';
  window.Superlists.initialize(url);
  server.respondWith('POST', url, [
    400,
    {"Content-Type": "application/json"},
    JSON.stringify({'error': 'something is amiss'})
  ]);
  $('.has-error').hide();

  $('#id_item_form').submit();
  server.respond(); // post

  assert.equal($('.has-error').is(':visible'), true);
  assert.equal($('.has-error .help-block').text(), 'something is amiss');
});

QUnit.test("should hide errors on post success", function (assert) {
    [...]
```

此外，你还要修改 base.html 模板，让它既能显示 Django 错误（主页现在就能显示），也能显示 JavaScript 错误：

lists/templates/base.html (ch36l031)

```
@@ -51,17 +51,21 @@
         <div class="col-md-6 col-md-offset-3 jumbotron">
           <div class="text-center">
             <h1>{% block header_text %}{% endblock %}</h1>
+
             {% block list_form %}
               <form id="id_item_form" method="POST" action="{% block [...]
                 {{ form.text }}
                 {% csrf_token %}
-                {% if form.errors %}
-                  <div class="form-group has-error">
-                    <div class="help-block">{{ form.text.errors }}</div>
+                <div class="form-group has-error">
+                  <div class="help-block">
+                    {% if form.errors %}
+                      {{ form.text.errors }}
+                    {% endif %}
                   </div>
-                {% endif %}
+                </div>
               </form>
             {% endblock %}
+
```

```
          </div>
        </div>
      </div>
```

最终，运行 JavaScript 测试应该得到类似下面的结果：

```
20 assertions of 20 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should call updateItems on initialize (1)
4. updateItems should get correct url by ajax (3)
5. updateItems should fill in lists table from ajax response (3)
6. should intercept form submit and do ajax post (4)
7. should call updateItems after successful post (1)
8. should not intercept form submit if no api url passed in (1)
9. should display errors on post failure (2)
10. should hide errors on post success (1)
11. should display generic error if no error json (2)
```

全部测试应该都能通过，包括所有功能测试：

```
$ python manage.py test
[...]
Ran 81 tests in 62.029s
OK
```

太棒了！！！

这就是我们自己动手使用 Django 构建的 REST API。如果需要提示，可以查看代码示例仓库中的代码（https://github.com/hjwp/book-example/tree/appendix_rest_api）。

不过，我不建议使用 Django 自己动手构建 REST API，在此之前，你至少应该考察一下 Django-Rest-Framework，详情参见附录 G。继续前行吧！

REST API 小贴士

- 不要重复编写 URL
  与面向浏览器的应用相比，API 的 URL 更为重要。尽量减少在测试中硬编码 URL 的次数。
- 不要直接处理原始 JSON 字符串
  让 `json.loads` 和 `json.dumps` 常伴你左右。
- JavaScript 测试应该使用 Ajax 模拟库
  Sinon 不错。Jasmine 自带了，Angular 也是。
- 牢记优雅降级和渐进增强
  尤其是把静态网站变成由 JavaScript 驱动的网站时，至少要让网站的核心功能在没有 JavaScript 时依然能使用。

附录G

Django-Rest-Framework

在附录 F 中，我们自己动手构建了一个 REST API。现在，来看看 Django-Rest-Framework，这是很多 Python/Django 开发者构建 API 时的首选工具。Django 的目的是为构建数据库驱动的网站提供各种基础工具（ORM、模板等），而 Django-Rest-Framework 的目的则是为构建 API 提供全部工具，从而避免一次次地编写样板代码。

撰写本附录时，我苦苦思索，怎样使用 Django-Rest-Framework 构建一个与前面自己动手实现的那个一模一样的 API 呢？若想在 Django-Rest-Framework 中得到同样的 URL 结构和 JSON 数据结构，面临巨大的挑战。在实现的过程中，我感觉自己是在与 Django-Rest-Framework 做斗争。

这提醒了我，让我陷入沉思。构建 Django-Rest-Framework 的人比我聪明得多，他们见过的 REST API 也比我多得多。如果他们觉得应该以某种方式处理，或许就表明我应该采用那种方式。我应该站在他们的角度思考问题，而不应该固执己见。

“别与框架做斗争”，这是我听过的至理名言之一。如果不能顺应框架，可能就要想想自己到底需不需要使用框架。

我们将以附录 F 构建的 API 为蓝本，尝试使用 Django-Rest-Framework 重写。

G.1 安装

Django-Rest-Framework 使用 `pip install` 命令就能安装。我使用的是创作本书时的最新版——3.5.4 版：

```
$ pip install djangorestframework
```

然后把 `rest_framework` 添加到 settings.py 中的 `INSTALLED_APPS` 设置中：

superlists/settings.py

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
    'accounts',
    'functional_tests',
    'rest_framework',
]
```

G.2 串化器（具体而言是`ModelSerializer`）

Django-Rest-Framework 官网的教程是学习这个框架的好资源。一开始你就会遇到串化器（serializer），这里具体而言是 `ModelSerializer`。Django-Rest-Framework 通过串化器把 Django 数据库模型转换为交换数据所需的 JSON（或其他格式）。

lists/api.py (ch37l003)

```
from lists.models import List, Item
[...]
from rest_framework import routers, serializers, viewsets


class ItemSerializer(serializers.ModelSerializer):

    class Meta:
        model = Item
        fields = ('id', 'text')


class ListSerializer(serializers.ModelSerializer):
    items = ItemSerializer(many=True, source='item_set')

    class Meta:
        model = List
        fields = ('id', 'items',)
```

G.3 Viewset（具体而言是`ModelViewSet`）和路由器

Django-Rest-Framework 使用 `ModelViewSet` 定义通过 API 与某个模型对象的交互方式。我们只需指明想操作的是哪个模型（通过 `queryset` 属性），以及如何序列化模型对象（`serializer_class`），余下的工作由 `ModelViewSet` 自动完成，即自动构建相关视图，供列出、获取、更新，甚至是删除对象。

为了从特定的清单中获取待办事项，只需定义这样一个 ViewSet：

lists/api.py (ch37l004)

```
class ListViewSet(viewsets.ModelViewSet):
    queryset = List.objects.all()
    serializer_class = ListSerializer


router = routers.SimpleRouter()
router.register(r'lists', ListViewSet)
```

Django-Rest-Framework 通过**路由器**自动构建 URL 配置，并把它们映射到 ViewSet 提供的功能上。

现在，我们可以修改 urls.py，绕开旧的 API 代码，指向新的路由器，看看测试的情况如何：

superlists/urls.py (ch37l005)

```
[...]
# from lists.api import urls as api_urls
from lists.api import router

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)),
    url(r'^accounts/', include(accounts_urls)),
    # url(r'^api/', include(api_urls)),
    url(r'^api/', include(router.urls)),
]
```

结果好多测试都失败了：

```
$ python manage.py test lists
[...]
django.urls.exceptions.NoReverseMatch: Reverse for 'api_list' not found.
'api_list' is not a valid view function or pattern name.
[...]
AssertionError: 405 != 400
[...]
AssertionError: {'id': 2, 'items': [{'id': 2, 'text': 'item 1'}, {'id': 3,
'text': 'item 2'}]} != [{'id': 2, 'text': 'item 1'}, {'id': 3, 'text': 'item
2'}]

 ---------------------------------------------------------------------
Ran 54 tests in 0.243s

FAILED (failures=4, errors=10)
```

先看那 10 个错误，报错消息都说无法反转 `api_list`。这是因为 Django-Rest-Framework 路由器使用的命名约定与前面我们自己制定的不同。从调用跟踪可以看出，这些错误发生在渲染模板时。具体而言，是 list.html 模板。我们只需修改一处就能修正这些错误——把 `api_list` 改成 `list-detail`：

lists/templates/list.html (ch37l006)

```
  <script>
$(document).ready(function () {
  var url = "{% url 'list-detail' list.id %}";
});
  </script>
```

这样修改之后，只剩下 4 个失败了：

```
$ python manage.py test lists
[...]
FAIL: test_POSTing_a_new_item (lists.tests.test_api.ListAPITest)
[...]
FAIL: test_duplicate_items_error (lists.tests.test_api.ListAPITest)
[...]
FAIL: test_for_invalid_input_returns_error_code
(lists.tests.test_api.ListAPITest)
[...]
FAIL: test_get_returns_items_for_correct_list
(lists.tests.test_api.ListAPITest)
[...]
FAILED (failures=4)
```

暂且关闭所有验证测试，后面再想办法解决：

lists/tests/test_api.py (ch37l007)

```
[...]
    def DONTtest_for_invalid_input_nothing_saved_to_db(self):
        [...]
    def DONTtest_for_invalid_input_returns_error_code(self):
        [...]
    def DONTtest_duplicate_items_error(self):
        [...]
```

现在只有 2 个失败了：

```
FAIL: test_POSTing_a_new_item (lists.tests.test_api.ListAPITest)
[...]
    self.assertEqual(response.status_code, 201)
AssertionError: 405 != 201
[...]
FAIL: test_get_returns_items_for_correct_list
(lists.tests.test_api.ListAPITest)
[...]
AssertionError: {'id': 2, 'items': [{'id': 2, 'text': 'item 1'}, {'id': 3,
'text': 'item 2'}]} != [{'id': 2, 'text': 'item 1'}, {'id': 3, 'text': 'item
2'}]
[...]
FAILED (failures=2)
```

先看最后 1 个失败。

Django-Rest-Framework 的默认配置得到的数据结构与我们自己动手构建时稍有不同，GET 请求清单得到的响应中有清单的 ID，而且清单中的待办事项在 `items` 键名下。因此，为了

让测试通过，我们要稍微修改一下单元测试：

lists/tests/test_api.py (ch37l008)

```
@@ -23,10 +23,10 @@ class ListAPITest(TestCase):
         response = self.client.get(self.base_url.format(our_list.id))
         self.assertEqual(
             json.loads(response.content.decode('utf8')),
-            [
+            {'id': our_list.id, 'items': [
                 {'id': item1.id, 'text': item1.text},
                 {'id': item2.id, 'text': item2.text},
-            ]
+            ]}
         )
```

现在能通过 GET 请求获取清单中的待办事项了（稍后将看到，随之一起返回的还有很多其他数据），那通过 POST 请求添加新待办事项呢？

G.4　通过POST请求添加待办事项的URL

这次我不再与框架做斗争了，而是顺应 Django-Rest-Framework。向清单的 ViewSet 发送 POST 请求是可以添加待办事项，但极为麻烦。

最简单的方法是向待办事项的 ViewSet 发送 POST 请求，而不是清单的 ViewSet：

lists/api.py (ch37l009)

```
class ItemViewSet(viewsets.ModelViewSet):
    serializer_class = ItemSerializer
    queryset = Item.objects.all()


[...]
router.register(r'items', ItemViewSet)
```

这意味着我们要稍微修改测试，把 POST 测试从 `ListAPITest` 中移出来，放到新的测试类 `ItemsAPITest` 中：

lists/tests/test_api.py (ch37l010)

```
@@ -1,3 +1,4 @@
 import json
+from django.core.urlresolvers import reverse
 from django.test import TestCase
 from lists.models import List, Item
@@ -31,9 +32,13 @@ class ListAPITest(TestCase):


+
+class ItemsAPITest(TestCase):
+    base_url = reverse('item-list')
+
     def test_POSTing_a_new_item(self):
```

```
        list_ = List.objects.create()
        response = self.client.post(
-           self.base_url.format(list_.id),
-           {'text': 'new item'},
+           self.base_url,
+           {'list': list_.id, 'text': 'new item'},
        )
        self.assertEqual(response.status_code, 201)
```

现在测试的结果为：

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

序列化待办事项时，如果没有指定清单的 ID，就不能知道待办事项属于哪个清单：

lists/api.py (ch37l011)

```
class ItemSerializer(serializers.ModelSerializer):

    class Meta:
        model = Item
        fields = ('id', 'list', 'text')
```

为此，还要修改另一个有点联系的测试：

lists/tests/test_api.py (ch37l012)

```
@@ -25,8 +25,8 @@ class ListAPITest(TestCase):
         self.assertEqual(
             json.loads(response.content.decode('utf8')),
             {'id': our_list.id, 'items': [
-                {'id': item1.id, 'text': item1.text},
-                {'id': item2.id, 'text': item2.text},
+                {'id': item1.id, 'list': our_list.id, 'text': item1.text},
+                {'id': item2.id, 'list': our_list.id, 'text': item2.text},
             ]}
         )
```

G.5 调整客户端代码

现在，这个 API 不再返回一个包含清单中所有待办事项的扁平数组，而是返回一个对象，待办事项都在它的 `.items` 属性中。因此，我们要稍微调整一下 `updateItems` 函数：

lists/static/list.js (ch37l013)

```
@@ -3,8 +3,8 @@ window.Superlists = {};
 window.Superlists.updateItems = function (url) {
   $.get(url).done(function (response) {
     var rows = '';
-    for (var i=0; i<response.length; i++) {
-      var item = response[i];
+    for (var i=0; i<response.items.length; i++) {
+      var item = response.items[i];
       rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
     }
     $('#id_list_table').html(rows);
```

而且，因为获取清单和添加待办事项的 URL 都变了，所以我们还要稍微调整一下 `initialize` 函数。我们不再使用多个参数，而是传入包含所需配置的 `params` 对象：

lists/static/list.js

```
@@ -11,23 +11,24 @@ window.Superlists.updateItems = function (url) {
   });
 };

-window.Superlists.initialize = function (url) {
+window.Superlists.initialize = function (params) {
   $('input[name="text"]').on('keypress', function () {
     $('.has-error').hide();
   });

-  if (url) {
-    window.Superlists.updateItems(url);
+  if (params) {
+    window.Superlists.updateItems(params.listApiUrl);

     var form = $('#id_item_form');
     form.on('submit', function(event) {
       event.preventDefault();
-      $.post(url, {
+      $.post(params.itemsApiUrl, {
+        'list': params.listId,
         'text': form.find('input[name="text"]').val(),
         'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
       }).done(function () {
         $('.has-error').hide();
-        window.Superlists.updateItems(url);
+        window.Superlists.updateItems(params.listApiUrl);
       }).fail(function (xhr) {
         $('.has-error').show();
         if (xhr.responseJSON && xhr.responseJSON.error) {
```

据此修改 list.html 中的代码：

lists/templates/list.html (ch37l014)

```
$(document).ready(function () {
  window.Superlists.initialize({
    listApiUrl: "{% url 'list-detail' list.id %}",
    itemsApiUrl: "{% url 'item-list' %}",
    listId: {{ list.id }},
  });
});
```

经过一番修改之后，基本的功能测试又能通过了：

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
Ran 2 tests in 15.635s

OK
```

为了解决前面暂时忽略的错误，还要做一些修改。如果不知道如何修改，可以参照本附录的仓库（https://github.com/hjwp/book-example/blob/appendix_DjangoRestFramework/lists/api. py）。

G.6 Django-Rest-Framework的优势

你可能想知道为什么要使用这个框架。

G.6.1 用配置代替代码

第一个优势是，以前的过程式视图函数变成了声明式句法：

lists/api.py

```
def list(request, list_id):
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
            return HttpResponse(status=201)
        else:
            return HttpResponse(
                json.dumps({'error': form.errors['text'][0]}),
                content_type='application/json',
                status=400
            )
    item_dicts = [
        {'id': item.id, 'text': item.text}
        for item in list_.item_set.all()
    ]
    return HttpResponse(
        json.dumps(item_dicts),
        content_type='application/json'
    )
```

如果与使用 Django-Rest-Framework 得到的最终版本相比，你会发现，我们完全是在配置：

lists/api.py

```
class ItemSerializer(serializers.ModelSerializer):
    text = serializers.CharField(
        allow_blank=False, error_messages={'blank': EMPTY_ITEM_ERROR}
    )

    class Meta:
        model = Item
        fields = ('id', 'list', 'text')
        validators = [
            UniqueTogetherValidator(
                queryset=Item.objects.all(),
```

```
                fields=('list', 'text'),
                message=DUPLICATE_ITEM_ERROR
            )
        ]


class ListSerializer(serializers.ModelSerializer):
    items = ItemSerializer(many=True, source='item_set')

    class Meta:
        model = List
        fields = ('id', 'items',)


class ListViewSet(viewsets.ModelViewSet):
    queryset = List.objects.all()
    serializer_class = ListSerializer


class ItemViewSet(viewsets.ModelViewSet):
    serializer_class = ItemSerializer
    queryset = Item.objects.all()


router = routers.SimpleRouter()
router.register(r'lists', ListViewSet)
router.register(r'items', ItemViewSet)
```

G.6.2 自带的功能

第二个优势是，使用 Django-Rest-Framework 的 `ModelSerializer`、ViewSet 和路由器得到的 API 比我们自己动手构建的 API 更具扩展性。

- 现在，清单和待办事项相关的所有 URL 都自动支持全部 HTTP 方法，包括 GET、POST、PUT、PATCH、DELETE 和 OPTIONS。
- 而且在 http://localhost:8000/api/lists/ 和 http://localhost:8000/api/items 可以浏览自动生成的 API 文档（你可以自己试试，如图 G-1 所示）。

除此之外，Django-Rest-Framework 还有很多优势，详情参见文档（http://www.django-rest-framework.org/topics/documenting-your-api/#self-describing-apis）。不过这两个功能对 API 的用户而言是十分重要的。

综上，Django-Rest-Framework 是构建 API 的优秀工具，几乎能根据现有的模型结构自动生成 API。如果你使用 Django，在自己动手实现 API 之前绝对应该先考察一下 Django-Rest-Framework。

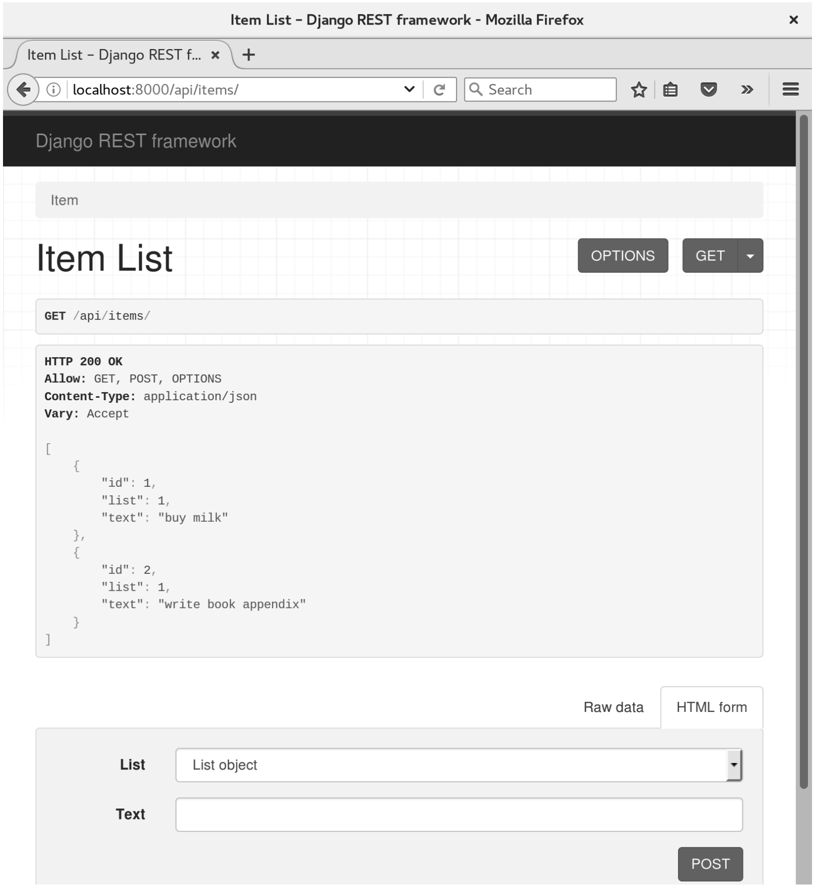


图 G-1：自动为 API 用户生成的文档

Django-Rest-Framework 小贴士

- 别与框架做斗争
  若想提高效率，通常最好顺应框架的约定，否则就不要使用框架，或者在较低的层级定制。
- 根据最小惊讶原则，使用路由器和 ViewSet
  Django-Rest-Framework 的优势之一是，使用它提供的工具（如路由器和 ViewSet）得到的 API 是可预料的，端点、URL 结构和不同 HTTP 方法的响应都有合理的默认配置。
- 查看可浏览的 API 文档
  在浏览器中访问 API 的端点。Django-Rest-Framework 能检测到你访问 API 时使用的是“常规的”Web 浏览器，此时它会显示自身的精美文档，可供你分享给你的用户。

附录H

速查表

人们都喜欢速查表，所以我根据每章末尾旁注中的总结制作了这个速查表，目的是提醒你，并且链接到具体章节，以此唤起你的记忆。希望这个速查表有用。

H.1　项目开始阶段

- 先构思一个**用户故事**，然后转换成第一个**功能测试**。
- 选择一个测试框架——`unittest` 不错，`py.test`、`nose` 和 `Green` 也有一定优势。
- 运行功能测试，得到第一个**预期失败**。
- 选择一个 Web 框架，例如 Django，然后弄清如何在选中的框架中运行**单元测试**。
- 针对目前失败的功能测试编写第一个**单元测试**，看着它失败。
- 做**第一次提交**，把代码提交到 VCS（例如 Git）中。

相关内容：第 1、2、3 章。

H.2　TDD基本流程

- 双循环 TDD（图 H-1）。
- 遇红 / 变绿 / 重构。
- 三角法。
- 便签。
- “三则重构”原则。
- “从一个可运行状态到另一个可运行状态”。
- “YAGNI”原则。

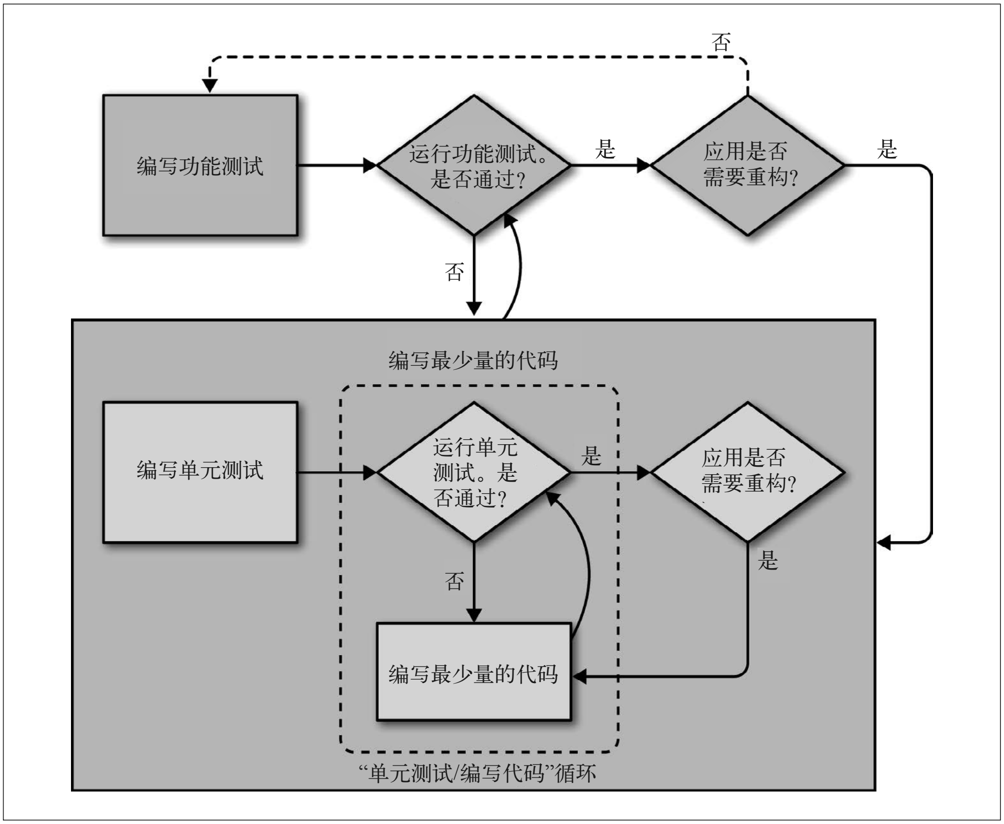


图 H-1：包含功能测试和单元测试的 TDD 流程

相关内容：第 4、5、7 章。

H.3 测试不止要在开发环境中运行

- 尽早进行系统测试。确保各组件能正常协作，包括 Web 组件、静态内容和数据库。
- 搭建和生产环境一样的过渡环境，在这个环境中运行功能测试。
- 自动部署过渡环境和生产环境：
  - PaaS 与 VPS；
  - Fabric；
  - 配置管理工具（Chef、Puppet、Salt 和 Ansible）；
  - Vagrant。
- 彻底弄清楚部署的主要步骤：数据库、静态文件、依赖、如何定制设定，等等。
- 尽早搭建 CI 服务器，运行测试不能只靠自律。

相关内容：第 9、11、24 章，附录 C。

H.4 通用的测试最佳实践

- 每个测试只能测试一件事。
- 应用的一个源码文件对应一个测试文件。
- 不管函数和类多么简单，都至少要编写一个占位测试。
- “别测试常量”。
- 尝试测试行为，而不是实现方式。
- 不能顺着代码的逻辑思考，还要考虑边缘情况和有错误的情况。

相关内容：第 4、13、14 章。

H.5 Selenium/功能测试最佳实践

- 相较隐式等待，多使用显式等待和交互等待模式。
- 避免编写重复的测试代码——可以在基类中定义辅助方法，也可以使用页面模式。
- 避免重复测试同一个功能。如果测试中有耗时操作（例如登录），可以找一种方法在其他测试中跳过这一步（但要小心看起来无关的功能相互之间的异常交互）。
- 使用 BDD 工具，作为组织功能测试的另一种方式。

相关内容：第 21、24、25 章。

H.6 由外而内，测试隔离与整合测试，模拟技术

别忘了编写测试的初衷。

- 确保正确性，避免回归。
- 有利于写出简洁可维护的代码。
- 实现一种快速高效的工作流程。

记住这几点之后，再看不同的测试类型以及各自的优缺点。

- *功能测试*
    - 从用户的角度出发，最大程度上保证应用可以正常运行。
    - 但反馈循环用时长。
    - 而且无法帮助我们写出简洁的代码。
- *整合测试（依赖于 ORM 或 Django 测试客户端等）*
    - 编写速度快。
    - 易于理解。
    - 发现任何集成问题都会提醒你。
    - 但并不总能得到好的设计（这取决于你自己）。
    - 而且一般运行速度比隔离测试慢。

- 隔离测试（使用驭件）
    - 涉及的工作量最大。
    - 可能难以阅读和理解。
    - 但这种测试最能引导你实现更好的设计。
    - 而且运行速度最快。

如果你发现编写测试时要使用很多驭件，而且感觉很痛苦，那么记得要**“倾听测试的心声”**——使用模拟技术写出的丑陋测试试图告诉你，代码可以简化。

相关内容：第 22、23、26 章。

附录I

接下来做什么

下面是我建议你接下来可以研究的一些事情，目的是提升测试技能，以及把（写作本书时的）新技术应用到 Web 开发中。

如果以后不再添加附录，我希望至少为每个话题写一篇博客文章，也编写一些示例代码。所以请读者一定要访问 http://www.obeythetestinggoat.com，看有没有更新。

或者你可以抢在我前面，自己写博客文章，记录你尝试其中任何一件事的过程。

我很乐意回答问题以及为这些话题提供提示和指引，所以如果你想尝试做某件事，但卡住了，别犹豫，联系我吧，电子邮件地址是 obeythetestinggoat@gmail.com。

I.1 提醒——站内提醒以及邮件提醒

如果有人把清单分享给某个用户，能提醒这个用户就好了。

你可以使用 django-notifications，在用户下次刷新页面时显示一个消息。在这个功能的功能测试中需要两个浏览器。

或者，也可以通过电子邮件提醒。研究一下 Django 对测试电子邮件的支持，然后你会发现，测试的过程需要发送真的电子邮件。使用 IMAPClient 库可以从网页邮件测试账户中获取真实的电子邮件。

I.2 换用Postgres

SQLite 对小型数据库来说很好，但如果有不止一个 Web 职程处理请求，那它就无法胜任了。现在，Postgres 是大家最喜欢的数据库，请弄清怎么安装及配置 Postgres。

你要找一个文件保存本地、过渡服务器和生产服务器中 Postgres 的用户名和密码。因为出于安全的考虑，你或许不想把这些信息放入代码仓库。你得找到一种方法，让部署脚本把这些信息传入命令行。流行的解决方法之一是在环境变量中存储这些信息。

你可以实验一下，看单元测试在 SQLite 中运行比在 Postgres 中运行快多少。为此，你可以在本地设备中使用 SQLite，仅做测试，但在 CI 服务器中使用 Postgres。

I.3 在不同的浏览器中运行测试

Selenium 支持各种浏览器，包括 Chrome 和 Internet Exploder。尝试在这两种浏览器中运行功能测试组件，看看有没有什么异常表现。

你还应该试试无界面浏览器，比如 PhantomJS。

根据我的经验，在不同的浏览器中测试能暴露 Selenium 测试中的各种条件竞争，而且可能还要更多地使用交互等待模式（尤其是在 PhantomJS 中）。

I.4 400和500测试

专业的网站需要漂亮的错误页面。400 页面的测试方法很简单，但如果想测试 500 页面，或许得编写一个故意抛出异常的视图。

I.5 Django管理后台

假设有个用户发邮件声称某个匿名清单是他的。为此，想实现一种手动解决方案，由网站的管理员在管理后台中手动修改记录。

弄清楚怎么启用和使用管理后台。编写一个功能测试，首先由一个未登录的普通用户创建一个清单，然后管理员登录，进入管理后台，把这个清单指派给这个用户，然后这个用户即可在“My Lists”页面看到这个清单。

I.6 编写一些安全测试

扩展针对登录、“My Lists”页面和分享功能的测试，看看需要怎么编写测试确保用户只能做有权限做的事情。

I.7 测试优雅降级

如果 Persona 不可用会发生什么？是否至少可以向用户显示一个致歉消息？

- 提示：模拟 Persona 服务不可用的方式之一是修改主机文件（路径是 /etc/hosts 或 c:\Windows\Sytem32\drivers\etc）。记得在测试的 `tearDown` 方法中撤销改动。
- 要同时考虑服务器端和客户端。

I.8 缓存和性能测试

弄清楚如何安装和配置 `memcached`，以及如何使用 Apache 的 `ab` 工具运行性能测试。在有缓存和没有缓存两种情况下，网站的性能如何？你能否编写一个自动化测试，如果检测到没启用缓存就失败？应该怎么处理可怕的缓存失效问题？测试能否帮你确认缓存失效逻辑是可靠的？

I.9 JavaScript MVC框架

现今，在客户端实现“模型－视图－控制器”（Model-View-Controller，MVC）模式的 JavaScript 库比较流行。这种库喜欢使用待办事项清单应用做演示，所以把这个网站改写成单页网站应该很容易。在单页网站中，添加清单的所有操作都由 JavaScript 代码完成。

选一个框架，Backbone.js 或 Angular.js，探究一下怎么实现。在各种框架中编写单元测试都有各自的方式。学习一种方式，一直使用下去，看你是否喜欢。

I.10 异步和websocket

假设两个用户同时编辑同一个清单，如果能看到实时更新，即一个用户添加待办事项之后，另一个用户立即就能看到，是不是很棒？这种功能可以通过使用 websocket 在客户端和服务器之间建立持久连接实现。

研究一种 Python 异步 Web 服务器，Tornado、gevent 或 Twisted，看你能否用它实现动态提醒。

测试时需要两个浏览器实例（就像在分享功能的测试中一样），检查不刷新页面的情况下，操作提醒是否会出现在另一个浏览器实例中。

I.11 换用py.test

使用 py.test 编写单元测试不用写那么多样板代码。尝试使用 py.test 改写一些单元测试。或许需要使用插件才能和 Django 无缝配合。

I.12 试试coverage.py

Ned Batchelder 开发的 coverage.py 能告诉你**测试的覆盖度**如何，即测试覆盖百分之多少的代码。目前，我们使用的是严格的 TDD，因此理论上覆盖度应该始终为 100%，不过再确认一下更好。而且对于没有从头开始编写测试的项目来说，这个工具是十分有用的。

I.13 客户端加密

这个比较有趣：如果用户太偏执，不再相信 NSA（美国国家安全局），觉得把清单放在云

端不安全该怎么办？你能不能使用 JavaScript 构建一个加密系统，在待办事项发给服务器之前，让用户输入密码，加密自己的清单。

针对这种功能的测试，可以这么写：管理员登录 Django 管理后台，查看用户的清单，确认清单中的待办事项在数据库中是否使用密文存储。

I.14 你的建议

你觉得我应该在这个附录中写些什么？提些建议吧！

附录 J

示例源码

本书的所有示例代码都在我的一个 GitHub 仓库中（https://github.com/hjwp/book-example/）。如果你想对比你我的代码，可以看一下那个仓库。

每一章都有自己的分支，分支名与章序一样，例如 chapter_01。

注意，各分支包含对应那一章的所有提交，因此是那一章**结束**时最终得到的代码。

此外，各章分别的代码示例也可至图灵社区下载，详情请见 http://www.ituring.com.cn/2052 “随书下载”处。

J.1 使用Git检查自己的进度

如果你想锻炼自己的 Git 技能，可以把我的仓库添加为**远程仓库**：

```
git remote add harry https://github.com/hjwp/book-example.git
git fetch harry
```

若想查看第 4 章**结束后**你我的代码有什么差异，可以这样做：

```
git diff harry/chapter_philosophy_and_refactoring
```

Git 能处理多个远程仓库，即便你已经把自己的代码推送到 GitHub 或 Bitbucket，依然可以这么做。

注意，类中方法的顺序在你我的代码中可能不完全相同，这可能导致差异不好读。

J.2 下载各章代码的ZIP文件

如果鉴于某些原因，阅读某一章时你想“从头做起”，或者跳过某一章，[1] 抑或你就是不想使用 Git，可以下载代码的 ZIP 文件，详情请见 http://www.ituring.com.cn/2052“随书下载”处。

J.3 不要完全依赖我的代码

除非真的卡住，不知道怎么做了，否则不要偷看答案。前面说过，自己动手调试错误能学到很多，而且当你自己开发时，可没有我的仓库供你对比，也没有现成的答案供你参考。

注 1：我不建议你跳着读。我在撰写时并没有考虑各章的独立性，后面的章节要依赖前面的章节，跳着读可能会更让你不明所以……

参考书目

[dip] Mark Pilgrim, *Dive Into Python*

[lpthw] Zed A. Shaw, *Learn Python The Hard Way*

[iwp] Al Sweigart, *Invent Your Own Computer Games With Python*

[tddbe] Kent Beck, *TDD By Example*, Addison-Wesley

[refactoring] Martin Fowler, *Refactoring*, Addison-Wesley

[seceng] Ross Anderson, *Security Engineering, Second Edition*, Addison-Wesley

[jsgoodparts] Douglas Crockford, *JavaScript: The Good Parts*，O’Reilly

[twoscoops] Daniel Greenfield and Audrey Roy, *Two Scoops of Django*

[mockfakestub] Emily Bache, *Mocks, Fakes and Stubs*

[GOOSGBT] Steve Freeman and Nat Pryce, *Growing Object-Oriented Software Guided by Tests*, Addison-Wesley

作者简介

Harry 的童年很美好，他在 Thomson T-07（当时在法国很流行，按键后会发出“啵噗”声）这种 8 位电脑上摆弄 BASIC，长大后做了几年经管顾问，但完全不快乐。而后他发现了自己真正的极客潜质，又很幸运地遇到了一些极限编程狂热者，参与开发了电子制表软件的先驱 Resolver One，不过很可惜，这个软件现在已经退出历史舞台。他目前在 PythonAnywhere LLP 公司工作，而且在各种演讲、研讨会和开发者大会上积极推广 TDD。

封面介绍

本书封面上的动物是开司米山羊。虽然所有山羊都长有开司米，但人类只选择培育这种山羊，产出能满足商用数量的开司米，所以一般只有这种山羊叫“开司米山羊”。因此，开司米山羊是一种驯养的家山羊。

开司米山羊长有一层异常柔软顺滑的内层绒毛，外覆一层粗糙的羊毛——这就是山羊的两层羊毛。开司米在冬季长成，目的是补充外层羊毛（这种毛叫“针毛”）的御寒能力。开司米中毛发的卷曲量决定了它的重量和保暖性能。

“开司米”这个名字出自印度次大陆上的克什米尔山谷地区。在这一地区，纺织品已经出现几千年了。现在的克什米尔地区，开司米山羊数量不断减少，所以不再出口开司米纤维。现在，大多数开司米毛织品都出自阿富汗、伊朗、蒙古国和印度，以及占主导地位的中国。

开司米山羊的羊毛有多种颜色和颜色搭配。雄性和雌性都长有犄角，夏季可用于散热，干农活时主人也能用它们更好地控制其他山羊。

封面图片出自 Wood 的 *Animate Creation* 一书。

O'REILLY®

Python测试驱动开发

本书手把手教你从头开发一个真正的Web应用，演示使用Python做测试驱动开发（TDD）的优势。你将学会如何在开发应用的每一个组成部分之前编写和运行测试，然后再编写最少量的代码让测试通过，最终得到简洁可用的代码。此外，你还会了解Django、Selenium、Git、jQuery和Mock的基础知识，以及其他目前流行的Web开发技术。

- 深入分析TDD流程，包括“单元测试/编写代码”循环和重构
- 使用单元测试检查类和函数，使用功能测试检查浏览器中的用户交互
- 学习何时、如何使用驭件，以及隔离测试和整合测试的优缺点
- 在过渡服务器中测试和自动部署
- 测试网站中集成的第三方插件
- 使用持续集成环境自动运行测试
- 使用TDD构建一个具有Ajax前端界面的REST API

哈利·J.W. 帕西瓦尔（Harry J.W. Percival），TDD积极践行者，曾参与开发电子制作表软件先驱Resolver One；目前就职于PythonAnywhere公司，经常受邀参加TDD和Python开发主题演讲、研讨会和开发者大会；取得了利物浦大学计算机科学硕士学位和剑桥大学哲学硕士学位。

“要使开发者保持头脑清醒，测试可谓至关重要。哈利完成了一项不可思议的工作，他不仅吸引了我们对测试的关注，而且还探索了切实可行的测试实践方案。”

——**Michael Foord**

Python核心开发者、unittest维护者

“这本书远不只是介绍了测试驱动开发，它还是一套完整的最佳实践速成课程，完整介绍了如何使用Python开发现代Web应用。”

——**Kenneth Reitz**

Python软件基金会特别会员

“真希望在我们学习Django时，就能有哈利的这本书。它对Django和多种测试实践进行了精彩讲解，难度恰当且不乏挑战性。”

——**Daniel Greenfeld和Audrey Roy**

*Two Scoops of Django*作者

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-48557-1

ISBN 978-7-115-48557-1
定价：119.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks